CEUR-WS.org/Vol-3893/Paper07.pdf

MethodProxies: A Safe and Fast Message-Passing Control Library

Sebastian Jordan Montaño[1], Juan Pablo Sandoval Alcocer[2], Guillermo Polito[1], Stéphane Ducasse[1] and Pablo Tesone[1]

[1] *Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, Park Plaza, Parc scientifique de la Haute-Borne, 40 Av. Halley Bât A, 59650 Villeneuve-d'Ascq, France*

[2] *Department of Computer Science, School of Engineering, Pontificia Universidad Católica de Chile, Santiago, Chile*

Abstract

The injection of monitoring code allows for real-time observation of the program, which has proven instrumental in developing tools that assist developers with various programming tasks. In dynamic languages such as Pharo, renowned for their rich meta-programming capabilities and dynamic method dispatch, such monitoring capabilities are particularly valuable. Message-passing control techniques are commonly used to monitor program execution at the method level, involving the execution of specific code before and after each method invocation. Implementing message-passing control techniques, however, poses many challenges, notably in terms of instrumentation overhead. Additionally, it is crucial for the message-passing mechanism to be safe: *i.e.,* to accommodate recursive and reflective scenarios to ensure that it does not alter the execution of the monitored program, which could potentially lead to infinite loops or other unintended consequences.

Over the years, numerous techniques have been proposed to optimize message-passing control. This paper introduces MethodProxies, a message-passing instrumentation library that offers minimal overhead and is safe. We conduct a comparison between MethodProxies and two commonly used techniques implemented in the Pharo programming language: method substitution using the `run:with:in:` hook and source code modification. Our results demonstrate that MethodProxies offers significantly lower overhead compared to the other two techniques and is safe against infinite recursion.

Keywords

instrumentation, message-passing control, error handling, method compilation

1. Introduction

In software development, a common challenge is understanding how a program behaves under various conditions during its execution. Without detailed insights, developers may struggle to pinpoint inefficiencies, identify bugs, or optimize performance. Code instrumentation serves as a solution to this problem by embedding additional code that allows for continuous tracking and analysis of program behavior in run time. This method not only aids in troubleshooting and refining software but also facilitates the creation of powerful development tools tailored to enhance overall software quality and functionality.

In pure object-oriented languages, such as Pharo or Smalltalk [1], objects communicate exclusively through message-passing: sending and receiving messages. Message-passing control involves managing this message-passing, typically by executing actions before or after a method's execution. These techniques are commonly utilized to monitor program execution at the method level, involving the execution of specific code before and after each method invocation [2, 3]. By integrating these control points, developers can seamlessly insert custom behaviors into the method execution cycle without altering the core logic of the methods themselves. Such techniques enable the collection and analysis of metrics that shed light on the application's execution flow and interactions, enhancing the understanding of the program's dynamic behavior.

IWST 2024: International Workshop on Smalltalk Technologies, July 8-11, 2024, Lille, France
sebastian.jordan@inria.fr (S. Jordan Montaño); juanpablo.sandoval@uc.cl (J. P. Sandoval Alcocer); guillermo.polito@inria.fr (G. Polito); stephane.ducasse@inria.fr (S. Ducasse); pablo.tesone@inria.fr (P. Tesone)
https://jordanmontt.fr (S. Jordan Montaño)

© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

The implementation of message-passing control techniques, however, introduces many challenges, particularly the overhead of instrumentation, which can potentially distort the collected data and massively degrade the application's performance. This creates a dilemma where the measurement process itself can impact the system behavior it aims to capture [4]. Additionally, it is crucial for the message-passing mechanism to be safe: *i.e.,* to accommodate recursive and reflective scenarios to ensure that it does not alter the execution of the monitored program, which could potentially lead to infinite loops or other unintended consequences [5, 6].

Over the years, numerous techniques have been proposed to optimize message-passing control such as source code modification, specialization of error handling, and exploitation of the virtual machine lookup algorithm, among others [2]. This paper introduces MethodProxies[1], a message-passing control instrumentation library that offers minimal overhead and is safe. This minimal overhead ensures that the message-passing control technique remains practical to use. In addition, MethodProxies have been designed to be safe and avoid falling in endless loops when controlling system features. It is available under the MIT open-source license.

We conduct an empirical evaluation through benchmarks in the Pharo programming language. We compare MethodProxies with two commonly used techniques: method substitution using the `run:with:in:`hook and source code modification. We describe how these techniques function and the considerations they employ to ensure safe message-passing control. Our results demonstrate that MethodProxies offers significantly lower overhead when compared to the other two techniques.

2. Safe Message Passing Control

When users want to control a method's execution, they typically desire to execute an action both before and after the method's execution. These actions are traditionally designated as the beforeMethod and the afterMethod, respectively [7, 8, 9]. A Meta-safe library designed for controlling message sending must meet the following requirements [5, 6].

We will explain each of them in detail below.

- **Meta-safe recursion.** The library should incorporate a meta-safe recursion mechanism to prevent infinite recursions when the instrumentation calls an instrumented method.
- **Thread safety.** There is a possibility of multiple threads calling the same instrumented method concurrently. To address this scenario, the library must handle meta executions in a thread-safe manner.
- **Safe handling of unwinding for exceptions and non-local returns.** Instrumented methods may encounter exceptions or non-local returns. The library should ensure the execution of the afterMethod irrespective of whether the instrumented method encounters an exception or initiates a non-local return.
- **Uninstrumentation.** Pharo is a live programming environment. For this reason, we need to uninstrument the code after the analysis to maintain the system's integrity. The library should restore the original methods after instrumentation.

2.1. Meta-safe recursion

The library should incorporate a mechanism to safeguard against infinite recursion when instrumented methods recursively invoke each other [5, 6]. This situation arises when users define the beforeMethod or afterMethod hooks and inadvertently include calls to other instrumented methods within these definitions. Such recursive calls lead to infinite loops due to repeated invocation of instrumented methods by the instrumentation itself.

[1]https://github.com/pharo-contributions/MethodProxies

Consider the Listing 1: A user instruments the method `AClass»foo`, producing the instrumented version outlined below:

```
AClass>>foo
    handler beforeMethod.
    "method code"
    handler afterMethod.

Handler>>before
    'foo method called' logMessage.
    anInstanceOfAClass foo.
```

Listing 1: Meta-recursive call example

In the beforeMethod definition, the handler logs a message and subsequently calls the same instrumented method `AClass»foo`. As a result, every execution of the instrumented method triggers the before action, which again invokes `AClass»foo`, leading to an infinite meta-recursion.

> **Requirement.** The library should provide a meta-safe recursion prevention mechanism to manage recursions originating from within the instrumentation code effectively.

2.2. Thread safety

There is a potential for multiple threads to invoke the same instrumented method concurrently. This scenario requires that the library handle meta-recursions appropriately, extending the consideration to multi-threaded environments. Consider this example: If one execution thread invokes the method `AClass»foo`, the beforeMethod will tag the execution as meta to prevent a recursive call to `AClass»foo`within the same thread. However, if another thread concurrently calls `AClass»foo`, the original meta tag does not affect this new invocation.

> **Requirement.** The library must manage meta-executions in a thread-specific manner. It should ensure that meta-executions are marked uniquely for each thread, allowing each thread's activities to be handled independently.

2.3. Safe handling of unwinding for exceptions and non-local returns

Instrumented methods may encounter exceptions or non-local returns, which can disrupt normal execution flow. A non-local return is a return instruction within a block closure that forces the return from the method where the block was defined. Non-local return instructions force an unwind of the call stack because it *jumps over* all the stack frames in between the block frame and its defining method frame. Non-local returns exist in programming languages such as Ruby, Scala, Kotlin, and Pharo, among others.

It is important for the library to guarantee the execution of the afterMethod regardless of whether the instrumented method encounters an exception or initiates a non-local return. Indeed, methods can experience non-local returns or raise exceptions that might abruptly terminate their execution, potentially preventing the afterMethod from being executed. Consider the example in Listing 2: A user instruments a method that includes a non-local return:

```
AClass>>foo
    handler before.
    condition ifAbsent: [ ^ self ].
    handler after.
```

Listing 2: Non-local return

In this scenario, if the condition specified by `ifAbsent:` is met, the method will exit prematurely, and the code following, including the afterMethod, will not execute. Therefore, it is essential for the library to ensure the afterMethod is always executed, maintaining a consistent and reliable execution flow, irrespective of exceptions or non-local returns.

Requirement. The library must ensure that the afterMethod executes under all circumstances, whether an exception occurs or a non-local return is initiated.

2.4. Uninstrumentation

All Pharo applications and tools co-exist in the same run-time environment. This means that instrumenting some code can affect other parts of the system that are not under analysis in unintended ways. One way to ensure the system's integrity is to remove the instrumentation after an analysis has been performed.

Requirement. The library must uninstrument all the methods that were instrumented, restoring them to their original state.

3. Current message passing control techniques

In this section, we will discuss two commonly used instrumentation techniques that users can employ to control message passing in Pharo. Note that not all the solutions presented in [2] are available today in Pharo.

3.1. Source code modification

A common instrumentation approach involves modifying the methods' source code to be instrumented. Given Pharo's fully reflective capabilities, users have the freedom to directly alter the source code of any method they wish to instrument [10]. However, this method places significant responsibility on the users to manage potential issues such as meta-recursions that may arise during the instrumentation process.

For example, consider the method before and after a source code instrumentation in Listing 3:

```
"Before Instrumentation"
AClass>>foo
    | temp1 |
    temp1 := self doSomething.
    ^ temp1

"After Instrumentation"
AClass>>foo
    self isMetaForActiveProcess ifFalse: [
    self runInMetaLevel: [ #beforeHandler ] ].
    [ | temp1 |
    temp1 := self doSomething.
    ^ temp1 ] ensure: [
        self isMetaForActiveProcess ifFalse: [
            self runInMetaLevel: [ #afterAction ] ] ]
```

Listing 3: Code before and after instrumentation

This implementation encapsulates the method to be instrumented within an `ensure:` block. This ensures that the afterMethod will be executed regardless of whether an exception occurs or a non-local return is initiated [11]. Additionally, we encapsulate the before and after actions to prevent their execution in the event of a meta-call.

3.2. run:with:in: method hook

In Pharo, the methods of a class are stored within a method dictionary. This dictionary forms an association between the method selector and the corresponding instance of the CompiledMethod class. Notably, methods in Pharo are ordinary objects and are instances of this CompiledMethod class.

Each time a message is sent in the Pharo environment, the Virtual Machine (VM) performs a lookup to find the compiled method corresponding to the selector. Once located, the VM executes this method on the receiver, passing the necessary arguments. If the object found in the method dictionary is not an instance of the CompiledMethod class, indicating an exceptional scenario, the Pharo VM addresses this by sending the special message `run:with:in:` to the found object. The `run:with:in:` method receives the method's selector, the arguments, and the receiver as parameters, allowing any class to implement it and thus manage method execution within the Pharo environment. This functionality is available by default in the standard Pharo's VM.

The `run:with:in:` technique replaces a compiled method instance in the method dictionary with an object understanding a `run:with:in:` message, referred to here as ProxyObject. This method is similar to the substitution technique described in [2], but with a critical distinction: the substituting object is not confined to instances of CompiledMethod. Instead, it can be an instance of any class, greatly expanding the possibilities for method substitution beyond traditional constraints. Typically, the CompiledMethod is replaced by a ProxyObject that encapsulates and preserves the original method. When `run:with:in:` is triggered, this ProxyObject may first execute a before action, then execute the original method, followed by an after action. The following Listing 4 provides an example of a `run:with:in:` method of a ProxyObject:

```
ProxyObject >> run: selector with: args in: aReceiver
    | v |
    self isMetaForActiveProcess ifFalse: [
        self runInMetaLevel: [ #beforeHandler] ].
    [
    v := originalMethod valueWithReceiver: aReceiver arguments: args
    ] ensure: [
        self isMetaForActiveProcess ifFalse: [
            self runInMetaLevel: [ #afterHandler ] ] ]
    ^ v
```

Listing 4: ProxyObject implementation of run:with:in:

Note that in our previous example, considerations for meta recursion, multi-threading, and local returns are also essential. Upon uninstrumentation, the original method can be restored simply by replacing it back into the method dictionary of the class. This procedure ensures that the integrity and functionality of the original method are maintained, even after modification and subsequent restoration.

It is important to note that this approach also involves at least two additional lookup executions: one for finding the implementor of `run:with:in:` and another for the implementor of `valueWithReceiver:arguments:`. This technique is not optimal for the Just-In-Time (JIT) compiler, as it should have an intermediate routine to box the arguments and massage the calls. Additionally, it is not favorable for inline caches because the methods stored in the methods dictionary are not actual methods. As a result, this technique has a performance drawback.

4. MethodProxies

MethodProxies is a method-based instrumentation library written in Pharo inspired by MethodWrappers [3] (see Section 7 for a comparison). It instruments Pharo code without specific virtual machine support. It permits the dynamic instrumentation of Pharo methods, enabling the execution of user-defined actions both before and after a method's execution. This functionality is achieved through two

method hooks: beforeMethod and afterMethod, which users are required to implement. These hooks are invoked whenever an instrumented method is called.

Our new implementation differs from this original work by stratifying the proxies in two parts: the trap and the handler. This design is meant to prevent user mistakes. The low-level concerns such as the code instrumentation and patching are defined by the framework itself. Users only need to define the beforeMethod and afterMethod hooks in a handler object. In addition, MethodProxies is safe:

MethodProxies has a robust architecture that enables method proxying without encountering infinite loops. To mitigate meta-recursions —when a user calls an instrumented method within another instrumented method in the same execution thread— MethodProxies employs a mechanism to determine the current execution level: whether it is at the meta-level or the base level. If the execution is identified as being at the meta-level, the beforeMethod and afterMethod hooks are bypassed, allowing execution to proceed normally as if no instrumentation were installed.

4.1. MethodProxies in a nutshell

Figure 1 illustrates the process of instrumenting the method `AClass»foo` using MethodProxies. The upper part of the figure shows the uninstrumented code. The class AClass has a method `foo` which is indirectly referred by a caller.

In the bottom part, the figure shows how the code looks with the instrumentation. The caller, instead of activating the original method `foo`, activates a `trapMethod` instead. This `trapMethod` activates the beforeMethod, the original method `foo`, and the afterMethod, respectively. The method object of the selector `#foo`, is replaced by this trap method, and the original method is hidden under a hidden selector named `__foo`.

- MethodProxies puts the instance of the compiled method `foo` under a hidden selector within the AClass method dictionary.
- It selects a prototype method with the same number of arguments as the method intended for instrumentation. Using literal patching in the prototype method, it integrates a call to the original method `foo` via the hidden selector. Furthermore, the prototype method incorporates calls to the before and after actions, the meta-safe mechanism, and the method deactivators.
- The instance of the compiled method associated with the selector `#foo` is replaced with the instrumented prototype method. As a result, when `AClass»foo`is called, it invokes the prototype method instead.
- During uninstrumentation, it restores the original method `foo`, which is hidden under the hidden selector.

MethodProxies offers a straightforward API that is simple to understand and use. Listing 5 presents a practical example demonstrating the API's usage:

```
"Define the handler and the proxy method"
handler :=  MpCountingHandler new.
proxy := MpMethodProxy
   onMethod: Object >> #error:
   handler: handler.
"Install the instrumentation"
proxy install.
proxy enableInstrumentation.
"Call the instrumented method"
1 error: 'foo'.
"Uninstrument"
proxy uninstall.
"Program's analysis"
```

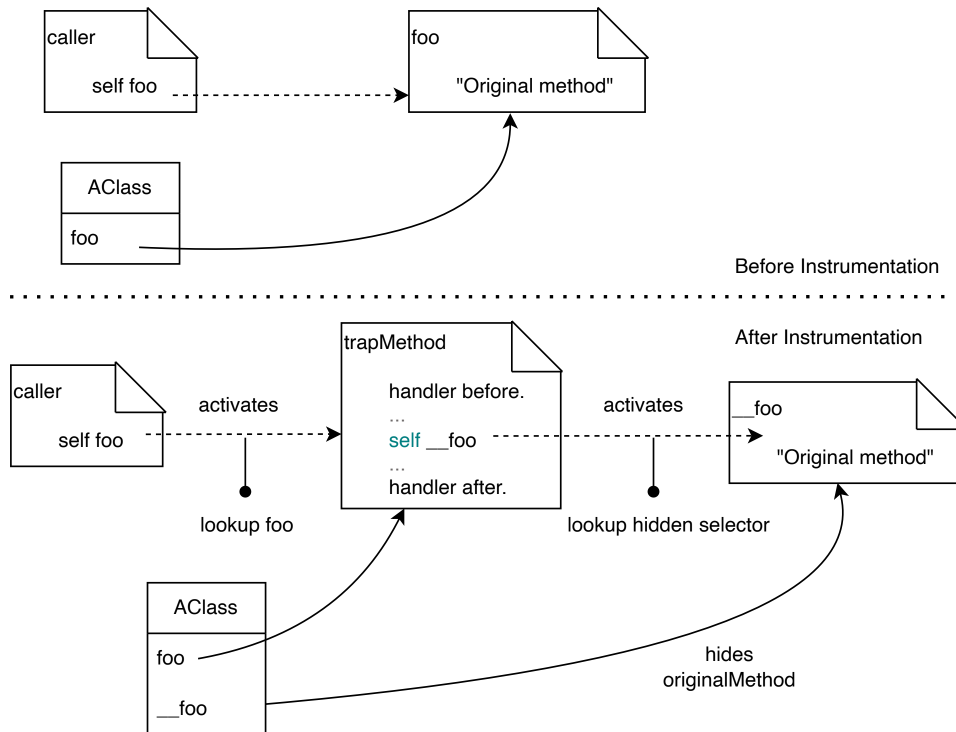

Figure 1: MethodProxies in a nutshell.

```
handler methodInvocations.
>>> 1
```

Listing 5: MethodProxies's API usage

4.2. The trap method

For MethodProxies, we introduced an alternative approach to implement the instrumentation: the trap method. During the instrumentation of the method, we copy a pre-compiled method template called the trap method. This trap method encapsulates the method intended for instrumentation alongside the beforeMethod and afterMethod hooks. Using literal patching, we modify the bytecode of our template trap method. Brant *et al.*, [3] use the same technique. It features a meta-safe mechanism to prevent the execution from entering into an infinite loop. Additionally, it incorporates deactivators that are triggered if the method intended for instrumentation raises an exception or has a non-local return. We hide the original method in the method dictionary under a hidden selector, to be able to restore it at a later stage.

```
AClass>> foo: args
    "This is not a primitive, just a marker"
    <primitive: 198>
    "The unwind handler should be the first temp.
    The complete flag should be the second temp."
    | deactivator complete result |
    deactivator := #deactivator.
    #beforeHandler.
    result := self ___foo: args.
    #afterHandler.
    "Mark the execution as complete to avoid double
    execution of the unwind handler"
    complete := true.
```

```
    ^ result
```

Listing 6: Trap method

```
AClass>> ___foo: args
    "Original source code hidden under the hidden selector"
```

Listing 7: Hidden original source code

Note that the after handler is not enclosed within an ensure block. This is unnecessary as we leverage the exception handling mechanism in Pharo to handle the eventual non-local returns and exceptions. Further details on this will be provided in the subsequent section.

4.3. Stack unwinding

A method may encounter non-local returns, causing the execution to jump to the frame where the non-local return was defined. This disrupts the flow of the trap method. If an exception is raised, it disrupts the execution in a similar manner.

To address this issue, one common approach is to utilize the `ensure:` method [12, 11]. The method `ensure:` expects a block as an argument and ensures its execution regardless of whether an exception or non-local return occurs. However, employing `ensure:` incurs a performance cost because it requires wrapping the code of the method in a block closure. This drawback leads us to opt for a different technique instead.

We introduced a technique that involves handling the stack unwinding within the trap method. This approach operates similarly to the `ensure:` method, but instead of encapsulating the code we want to ensure its execution, we embed the code directly within the trap method. To implement this, we annotate the method with a special primitive designed to always fail, thus marking it for stack unwinding. The first temporary variable is designated to store the unwind block that needs execution. Despite being labeled a primitive, this construct is not an actual primitive because it is intended to fail consistently. Its sole purpose is to indicate, during stack unwinding, the methods in which Pharo must execute the unwind block, mirroring the functionality of the `ensure:` method. Additionally, we implement a method deactivator, which is a specialized object responsible for executing the afterMethod if an exception is raised.

In employing this technique, we initially execute the code as usual, presuming no exceptions will be raised. If an exception or non-local return does occur, we activate the deactivators using the same exception handling mechanism as the `ensure:` method.

By marking the trap method with a marker primitive, Pharo's exception mechanism triggers the execution of our deactivators if an exception occurs. This technique mitigates the performance cost because no block closures are created. To achieve this optimization, we made changes to the way Pharo handles exceptions. These enhancements have been integrated into Pharo 12.

We will take the code snippet in Listing 6 as an example to explain the method deactivators. If, during the execution of the trap method installed in `AClass»foo`, an exception or a non-local return is encountered, then:

- Pharo's exception mechanism will treat the first temporary variable as the unwind block.
- Next, Pharo's exception mechanism will check the second temporary variable. If its value is not true, then it will execute the unwind block.

If there are no exceptions or non-local returns during execution, the deactivator will not be executed, and the execution will proceed normally.

5. Experimental setup

We designed an experiment to contrast MethodProxies against `run:with:in:`and source code modification techniques. Our goal is to understand the impact of MethodProxies in terms of instrumentation time, overhead, and uninstrumentation time. All the experimentation code is available online[2].

5.1. Research questions

This paper studies the following research questions regarding instrumentation techniques in Pharo:

- **RQ1 - Instrumentation and uninstrumentation overhead:** *How does the instrumentation time of MethodProxies compare to the `run:with:in:`and the source code modification instrumentation techniques?* This question aims to understand the impact of MethodProxies on instrumentation and uninstrumentation time.
- **RQ2 - Execution overhead:** *How does the overhead of MethodProxies compare to the `run:with:in:`and the source code modification instrumentation techniques?* This question aims to compare the overhead time of MethodProxies with that of `run:with:in:`and source code modification techniques. We aim to assess the impact of the improvements in terms of execution time overhead.

5.2. Projects under analysis

Table 1 describes the four projects we used for our analysis. It also reports the number of methods to instrument and the number of tests. To execute these projects, we run all its associated tests. We define a **benchmark** as the execution of a project's test suites.

Table 1
Projects under analysis

Project's name	Description	# methods	# tests
Compression	A package that provides compression utilities, including functionalities for compressing and decompressing files using ZIP and GZIP formats.	387	29
File System Manager	This project encompasses Pharo's file system, disk, path, and memory files manager functionalities.	1426	450
Microdow	Microdown[13] is a markup language based on Markdown, offering flexible extension mechanisms for creating books, slides, and websites.	1041	472
AST	This package contains the model for the abstract syntax tree (AST) representation available in the Pharo image.	1591	641

5.3. Scenarios under study

Given that the overhead may vary with the type of information collected during execution, we have developed three analysis tools:

- **Method call graph:** This analysis tool instruments all methods within an application to generate a graph that illustrates the relationships and frequencies of method calls. The method call graph is constructed with method-level granularity, with each method call being counted as an execution. It reports the frequency of method calls and identifies the callers. Additionally, the method call graph accommodates multi-threaded executions by distinguishing methods called from threads other than the executing one. Each time an instrumented method is called, the analysis updates the method call graph.

[2]https://github.com/jordanmontt/pharo-instrumentation

- **Method coverage:** This tool instruments all methods within an application to record which methods are invoked by a set of tests. It identifies methods that were invoked, as well as those that were not, during the test execution. The coverage tool operates at method granularity, considering a method executed if it is called during testing. Each time an instrumented method is called, the tool marks the method as executed in a table.
- **No-action instrumentation:** This tool involves instrumenting all methods within an application without executing any actions. It utilizes empty method bodies for both the beforeMethod and afterMethod hooks. This setup allows for the evaluation of the overhead associated with bare instrumentation.

5.4. Techniques under analysis

We employed three distinct instrumentation techniques— (i) MethodProxies, (ii) `run:with:in:`hook, and (iii) source code modification. Despite the variation in these techniques, we ensured uniform implementation for both the method call graph and method coverage across all approaches. Each analysis tool was implemented in a manner agnostic to the specific instrumentation techniques used. This uniform approach allows us to deploy the same analysis tools across all instrumentation techniques, such as MethodProxies, `run:with:in:`hook, and source code modification, thus introducing consistent overhead across techniques. Additionally, we applied the same meta-checking mechanism across all instrumentation techniques to maintain consistent meta-safety overhead among them.

It is important to note that the analysis tools themselves introduce varying levels of overhead. For instance, the method call graph tool requires additional calculations to determine the relationships between callers and callees, while the method coverage tool simply marks the executed methods. This variation in computational workload may lead to different overhead impacts for the same projects.

5.5. Benchmark execution and metrics

To ensure accurate measurement of execution times, we take the following considerations:

- We performed 30 VM iterations for each benchmark, with a single VM invocation per iteration, consistent with recommendations from prior studies [14]. We utilized ReBench [15] to manage our benchmark execution efficiently.
- The Pharo VM uses a non-optimizing baseline JIT compiler that compiles methods upon their second invocation and does not apply further optimizations. Thus, we contend that limiting our benchmarks to one iteration per VM invocation does not impact our results adversely.
- We calculated and reported both the mean and the standard deviation for each benchmark's measurements. To reduce system-related noise during benchmark executions, we terminated all non-essential OS applications and disabled the internet connection.

We consider the execution of all tests within each project as benchmarks for our measurements. For each analysis tool and project, we collected the following metrics:

- **Overhead Time:** This metric is calculated as the ratio between the execution time with instrumentation (*I*) and the execution time without instrumentation (*NI*).

$$Overhead = I \;/\; NI \tag{1}$$

- **Instrumentation overhead:** This represents the duration required by the analysis tools to instrument all methods prior to executing the benchmarks. It is calculated as the ratio between the instrumentation time (*insTime*) and the lowest instrumentation execution time (*lowInsTime*).

$$Instrumentation\ Overhead = insTime \;/\; lowInsTime. \tag{2}$$

- **Uninstrumentation overhead:** This measures the time taken by the analysis tools to restore the original compiled methods in the class. Similarly, it is calculated as the ratio between the uninstrumentation time (*uninsTime*) and the lowest uninstrumentation execution time (*lowUninsTime*).

$$Uninstrumentation\ Overhead = uninsTime\ /\ lowUninsTime. \tag{3}$$

6. Results

In this section, we present the results of our experiments. For each execution time, we report the mean value with the standard deviation over 30 runs.

6.1. RQ1 - Instrumentation and uninstrumentation overhead

For the first research question, we analyze how much time instrumenting and uninstrumenting the methods takes.

Table 2 presents the results of the benchmarks for the instrumentation overhead. Results are presented relative to the shortest time: lower values indicate better performance. The `run:with:in:` technique exhibits the lowest instrumentation overhead, with the fastest time being 140 milliseconds for the Compression benchmark, which has 387 methods to be instrumented. The shortest execution time by scenario among all analysis tools is in bold.

The `run:with:in:` hook exhibits the least instrumentation overhead, ranging from 1.0 to 1.57 ×. This is expected since this technique involves replacing the instance of CompiledMethod with self.

The source code modification technique incurs the most overhead for instrumentation, ranging from 6.36 to 282.37 ×. This is due to the requirement of string concatenation for creating the new methods' source code and recompiling all methods The overhead introduced by MethodProxies ranges from 1.16 to 2.38 ×. MethodProxies takes an instance of a pre-compiled method and replaces its references.

Table 2
Instrumentation overhead in milliseconds

	MethodProxies	run:with:in:	Source code modification
	No-action tool		
FileSystem	2.13 × ±0.03	1.43 × ±0.01	20.78 × ±0.07
Microdown	1.64 × ±0.02	1.15 × ±0.01	12.46 × ±0.06
Compression	1.17 × ±0.03	**1.0 × ±0.0**	6.36 × ±0.0
AST	2.37 × ±0.03	1.53 × ±0.04	148.92 × ±2.02
	Call graph tool		
FileSystem	2.14 × ±0.0	1.43 × ±0.01	25.5 × ±0.12
Microdown	1.63 × ±0.02	1.16 × ±0.03	13.3 × ±0.08
Compression	1.17 × ±0.04	**1.0 × ±0.0**	6.65 × ±0.03
AST	2.38 × ±0.03	1.57 × ±0.0	282.37 × ±8.68
	Method coverage tool		
FileSystem	2.14 × ±0.0	1.43 × ±0.0	22.81 × ±0.1
Microdown	1.65 × ±0.01	1.16 × ±0.03	13.04 × ±0.08
Compression	1.16 × ±0.03	**1.0 × ±0.0**	6.57 × ±0.02
AST	2.36 × ±0.01	1.56 × ±0.02	199.28 × ±3.67

Table 3 presents the benchmark results for the uninstrumentation overhead. The shortest execution is 140.0 ±0.0 milliseconds for the Compression benchmark. The largest uninstrumentation time for uninstrumenting all the methods is 280.0 ±0.0 milliseconds for the AST benchmark. We use the same

uninstrumentation mechanisms across all different analysis tools: we restore the compiled method object into the method dictionary. We will also present the numbers relative to the shortest execution time.

Table 3
Uninstrumentation overhead in milliseconds

	MethodProxies	**run:with:in:**	**Source code modification**
	No-action tool		
FileSystem	1.78 × ±0.01	1.14 × ±0.0	1.65 × ±0.02
Microdown	1.17 × ±0.04	1.08 × ±0.02	1.3 × ±0.03
Compression	**1.0** × **±0.0**	1.07 × ±0.0	1.07 × ±0.0
AST	1.92 × ±0.02	1.21 × ±0.0	1.93 × ±0.01
	Call graph tool		
FileSystem	1.86 × ±0.02	1.21 × ±0.0	1.58 × ±0.02
Microdown	1.29 × ±0.0	1.21 × ±0.0	1.35 × ±0.02
Compression	**1.07** × **±0.0**	1.15 × ±0.01	1.14 × ±0.02
AST	2.0 × ±0.0	1.29 × ±0.02	1.92 × ±0.02
	Method coverage tool		
FileSystem	1.85 × ±0.01	1.21 × ±0.0	1.5 × ±0.0
Microdown	1.24 × ±0.03	1.15 × ±0.02	1.27 × ±0.03
Compression	**1.07** × **±0.0**	1.14 × ±0.0	1.12 × ±0.03
AST	1.94 × ±0.02	1.29 × ±0.0	1.79 × ±0.01

RQ.1 How does the instrumentation time MethodProxies compare to other instrumentation techniques?
MethodProxies incurs an instrumentation overhead ranging from 1.16 to 2.38 × compared to the fastest time of `run:with:in:`. While MethodProxies has a higher overhead than `run:with:in:`, it is significantly lower than the source code modification technique. For uninstrumentation, MethodProxies exhibits the lowest overhead, with values ranging from 1.07 to 2.0 × across all analysis tools.

6.2. RQ2 - Execution overhead

For this research question, we first run the benchmarks without the instrumentation to calculate their baseline execution time. Then, we present the execution overhead relative to this baseline execution time.

6.2.1. Baseline execution time

To compare the overhead added by the instrumentation, we initially calculated the execution time of the benchmarks, which we call the baseline execution time. Table 4 presents the results for the execution time of the benchmarks without instrumentation. FileSystem tests are the ones that take the longest to execute, while the Microdown tests are the fastest.

Table 4
Baseline execution time in milliseconds (no instrumentation)

	FileSystem	**Microdown**	**Compression**	**AST**
Execution time	10788 ±27.21	1330 ±128.3	2575 ±11.37	4733 ±44.73

6.2.2. Execution overhead

Table 5 illustrates the execution overhead relative to the baseline execution time. We highlighted in bold the shortest execution time across all instrumentation techniques. MethodProxies has the lowest overhead among all benchmarks and all analysis tools. For the no-action tool, aimed at analyzing the cost of bare instrumentation, MethodProxies exhibits overheads between 0.91 and 5.15 $\times$ compared to the baseline execution. Interestingly, in the no-action tool with the Microdown benchmark, the execution was faster with MethodProxies than without instrumentation (0.91 $\times$), and we did not investigate further this.

On the contrary, `run:with:in:` demonstrates the highest execution overhead across all benchmarks and analysis tools. One of the reasons for this can be the additional lookup required by the VM to locate where `run:with:in:` is implemented, as well as its lack of compatibility with inline caches and the JIT compiler. Additionally, the AST benchmark displays a significant overhead across all analysis tools, which warrants further investigation.

The source code modification overheads are in the middle, between MethodProxies and `run:with:in:`. We inline the code of the beforeMethod and afterMethod but we wrap the afterMethod inside an `ensure:` block.

Table 5
Overhead for executing the instrumented code

	MethodProxies	run:with:in:	Source code modification
		No-action tool	
FileSystem	1.03 × ±0.0	1.17 × ±0.0	1.08 × ±0.0
Microdown	**0.91** × **±0.1**	17.98 × ±1.05	6.14 × ±0.16
Compression	1.05 × ±0.0	9.33 × ±0.22	3.31 × ±0.01
AST	5.15 × ±0.05	47.92 × ±2.75	23.33 × ±0.06
		Call graph tool	
FileSystem	**1.07** × **±0.0**	1.22 × ±0.0	1.11 × ±0.0
Microdown	4.35 × ±0.2	20.87 × ±1.13	8.7 × ±0.16
Compression	2.49 × ±0.01	10.56 × ±0.13	4.35 × ±0.01
AST	25.48 × ±0.23	49.87 × ±0.77	37.62 × ±0.15
		Method coverage tool	
FileSystem	**1.05** × **±0.0**	1.19 × ±0.0	1.09 × ±0.0
Microdown	2.22 × ±0.11	19.18 × ±0.8	6.82 × ±0.15
Compression	1.58 × ±0.01	9.73 × ±0.23	3.59 × ±0.01
AST	11.61 × ±0.13	44.08 × ±0.62	28.89 × ±0.09

RQ.2 How does the overhead of MethodProxies compare to the `run:with:in:` and the source code modification tools?
Among all benchmarks and analysis tools, MethodProxies exhibits the lowest execution overhead.

7. Related Work

In this section, we present the related work on instrumentation techniques and message-passing control.

Analysis tools. Bergel *et al.,* [16, 17] developed Spy, a profiling framework that allows developers to build custom profilers. Spy instruments the desired methods to execute actions before and after a method's execution. It uses the `run:with:in:` technique to instrument the code and control message passing. Spy has been used to build various tools, including method coverage tools [18], among others. In our work, we study different instrumentation techniques, such as MethodProxies, `run:with:in:`,

and source code modification, comparing the overhead of these techniques. Unlike Bergel *et al.,* which focus on the tools built using these instrumentation techniques, our focus is on the instrumentation techniques themselves and their associated overhead.

MethodWrappers. Brant *et al.,* [3] introduced Method Wrappers, a mechanism for introducing new behavior to be executed before or after a method. The authors explore several implementations of wrappers in Smalltalk and compare their performance with various program analysis tools, making this work the most similar to ours.

In their approach, they replace the instance of the CompiledMethod with an instance of a MethodWrapper. This new method includes the before action, the original method, and the after action, executing the original method using the `valueWithReceiver:arguments:` method. This technique does not add a new entry to the method dictionary and it does not hide the original method under a hidden selector. However, the MethodWrapper implementation lacks safety, requiring users to subclass MethodWrappers to handle the instrumentation themselves. This leaves the responsibility of managing safety mechanisms, such as meta-recursions, to the user.

In contrast, we implemented MethodProxies by replacing the instance of the compiled method with a pre-compiled trap method, which we then patch using literal patching. This approach is similar to MethodWrappers but offers a more robust and stratified architecture. With MethodProxies, users can define before and after methods without worrying about safety concerns such as meta-recursions. The user only needs to define the before and after methods, as the safety-ensuring mechanisms are handled automatically.

Sub-method Reflection. Reflectivity [19, 20, 21, 22] is a framework that allows developers to annotate abstract syntax tree (AST) nodes with meta-behavior, influencing the compiler to produce behavioral variations. These annotations are dynamically applied to AST nodes, which are then expanded, compiled, and executed. Notably, in Pharo, the AST is accessible at the language level, facilitating its modification. Reflectivity provides the essential infrastructure to support these capabilities. The front end of Reflectivity is designed to operate at AST level. We excluded Reflectivity from our comparison because is conceptually equivalent to the source code modification, as it needs to recompile the method with the AST modifications.

Infinite meta-recursions. Denker *et al.,* [6] worked on the problem of infinite meta-recursions in reflective languages. Mainstream languages use a reflective architecture to enable reflection. In this architecture, meta-objects control the different aspects of reflection offered by the language. The authors extended the meta-object-based reflective systems with a first-class notion of meta-level execution and the possibility to query at any point in the execution whether we are executing at the meta-level or not.

In CLOS, Kiczales *et al.,* [23] introduced an approach to programming language design, focusing on the evolution and principles of the Common Lisp Object System (CLOS) metaobject protocol. The work emphasizes that metaobject protocols enable users to customize programming languages to better meet their needs. The authors used memoization to speed up method lookup and dispatch.

Chiba *et al.,* [5] presented a new architecture, called the meta-helix, for systems that use the meta-object protocol. A common element of meta-object protocol design is the use of metacircularity to allow extensions to be implemented in terms of the original non-extended functionality. However, this design can lead to recursion due to the conflation of the extended and non-extended functionalities. Meta-helix architecture retains the benefits of metacircularity while addressing its problems.

We used these definitions of infinite meta-recursions as the foundation for building MethodProxies.

8. Discussion and future work

In this section, we will discuss the threats to validity and outline our future work.

Pharo's unsafe threads In Pharo, it is possible to terminate a thread at any point during its execution, even if the thread is being executed in a critical section. This capability contrasts with many mainstream languages, such as Java [3], which do not allow thread termination. This feature in Pharo can lead to significant issues. For instance, if a thread is executing code that marks the execution as being at the meta-level and another thread terminates it, the execution state will become inconsistent or corrupted. This situation falls outside the control of MethodProxies, as it is inherent to Pharo's threading model.

Special methods used by the instrumentation. MethodProxies employs some special methods to instrument the code. These special methods cannot be instrumented, as they are essential for the instrumentation process. To prevent their instrumentation, we mark these methods with the pragma <noInstrumentation>. These methods are specific to MethodProxies, so users typically will not need to instrument them.

Other instrumentation techniques. Our study compared MethodProxies against two commonly utilized message-passing control techniques: `run:with:in:` and source code modification. Although these techniques have provided valuable insights into instrumentation overheads, it is important to acknowledge the existence of other possible techniques outlined in the literature [2] that were not included in this comparison. As part of our future work, we plan to explore and analyze additional instrumentation techniques to deliver a more comprehensive comparison and deepen our understanding of their associated overheads.

Benchmarks choice. For our experiments, we opted to execute all tests within the selected benchmarks. While this approach ensures comprehensive coverage, it may not accurately represent these benchmarks' most typical use cases. In future research, we aim to select a set of benchmarks that do not rely solely on test execution, thereby providing a more representative evaluation of the instrumentation techniques.

Safe vs. unsafe profiling. Our experiments focused on comparing different techniques of safe message-passing control for developing analysis tools. However, we did not investigate the impact of the safe mechanism itself or the additional overhead it introduces. In some scenarios, users might not require the safe-checking mechanism. In future work, we plan to include a comparison of safe vs. unsafe profiling to gain a deeper understanding of the overhead introduced by the safe-checking mechanism.

9. Conclusion

This paper introduces MethodProxies, a new, safe, and fast message-passing instrumentation library tailored for the Pharo programming language. We demonstrate how MethodProxies efficiently handles multithreading, meta-recursions, exceptions, and local return scenarios. Furthermore, we present an experiment in which MethodProxies was assessed alongside two widely used techniques: the `run:with:in:` hook and source code modification, across a variety of profiling scenarios and projects. Our findings indicate that MethodProxies consistently exhibits the lowest execution overhead, significantly outperforming both the `run:with:in:` hook and source code modification in all evaluated benchmarks and analysis tools. By leveraging the exception mechanism in Pharo and avoiding block closures, MethodProxies delivers optimized performance.

The development of MethodProxies marks a significant advancement over traditional message-passing techniques. Looking ahead, future research will focus on further optimizations of MethodProxies, comparing it with additional instrumentation techniques, investigating various types of benchmarks, exploring the impact of safe vs. unsafe profiling, and extending its applicability to other dynamic programming languages and runtime environments.

[3] https://docs.oracle.com/javase/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html

Acknowledgments

We want to thank the Programa de Inserción Académica 2022, Vicerrectoría Académica y Prorrectoría, at the Pontificia Universidad Católica de Chile. We also extend our gratitude to the European Smalltalk User Group (ESUG) for financing part of this work.

References

[1] S. Ducasse, G. Rakic, S. Kaplar, Q. D. O. written by A. Black, S. Ducasse, O. Nierstrasz, D. P. with D. Cassou, M. Denker, Pharo 9 by Example, Book on Demand – Keepers of the lighthouse, 2022. URL: http://books.pharo.org.

[2] S. Ducasse, Evaluating message passing control techniques in Smalltalk, Journal of Object-Oriented Programming (JOOP) 12 (1999) 39–44.

[3] J. Brant, B. Foote, R. Johnson, D. Roberts, Wrappers to the rescue, in: Proceedings European Conference on Object Oriented Programming (ECOOP'98), volume 1445 of *LNCS*, Springer-Verlag, 1998, pp. 396–417.

[4] T. Mytkowicz, P. F. Sweeney, M. Hauswirth, A. Diwan, Observer effect and measurement bias in performance analysis, Computer Science Technical Reports CU-CS-1042-08, University of Colorado, Boulder (2008). URL: https://core.ac.uk/download/pdf/54846997.pdf.

[5] S. Chiba, G. Kiczales, J. Lamping, Avoiding confusion in metacircularity: The meta-helix, in: K. Futatsugi, S. Matsuoka (Eds.), Proceedings of ISOTAS '96, volume 1049, Springer, 1996, pp. 157–172. URL: http://www2.parc.com/csl/groups/sda/publications/papers/Chiba-ISOTAS96/for-web.pdf. doi:10.1007/3-540-60954-7_49.

[6] M. Denker, M. Suen, S. Ducasse, The meta in meta-object architectures, in: Proceedings of TOOLS EUROPE 2008, volume 11 of *LNBIP*, Springer-Verlag, 2008, pp. 218–237. doi:10.1007/978-3-540-69824-1_13.

[7] G. Attardi, C. Bonini, M. R. Boscotrecase, T. Flagella, M. Gaspari, Metalevel programming in CLOS, in: S. Cook (Ed.), Proceedings ECOOP '89, Cambridge University Press, Nottingham, 1989, pp. 243–256.

[8] G. Kiczales, J. des Rivières, D. G. Bobrow, The Art of the Metaobject Protocol, MIT Press, 1991.

[9] D. G. Bobrow, R. P. Gabriel, J. White, CLOS in context — the shape of the design, in: A. Paepcke (Ed.), Object-Oriented Programming: the CLOS perspective, MIT Press, 1993, pp. 29–61.

[10] I. Thomas, S. Ducasse, P. Tesone, G. Polito, Pharo: a reflective language - analyzing the reflective api and its internal dependencies, Journal of Computer Languages (2024). doi:10.1016/j.scico.2014.02.016.

[11] C. Bera, S. Ducasse, Handling exceptions, in: Deep Into Pharo, Square Bracket Associates, 2013, p. 38. URL: http://books.pharo.org.

[12] S. Costiou, T. Dupriez, D. Pollet, Handling Error-Handling Errors: dealing with debugger bugs in Pharo, in: International Workshop on Smalltalk Technologies - IWST 2020, 2020. URL: https://hal.inria.fr/hal-02992644.

[13] S. Ducasse, L. Dargaud, G. Polito, Microdown: a clean and extensible markup language to support pharo documentation, in: Proceedings of the 2020 International Workshop on Smalltalk Technologies, 2020.

[14] A. Georges, D. Buytaert, L. Eeckhout, Statistically rigorous java performance evaluation, in: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA '07, Association for Computing Machinery, New York, NY, USA, 2007, pp. 57–76. URL: https://doi.org/10.1145/1297027.1297033. doi:10.1145/1297027.1297033.

[15] S. Marr, Rebench: Execute and document benchmarks reproducibly, 2018. doi:10.5281/zenodo.1311762, version 1.0.

[16] A. Bergel, F. Bañados, R. Robbes, D. Röthlisberger, Spy: A flexible code profiling framework, in: Smalltalks 2010, 2010.

[17] A. Bergel, F. Banados, R. Robbes, D. Röthlisberger, Spy: A flexible code profiling framework, Journal of Computer Languages, Systems and Structures 38 (2011). URL: http://bergel.eu/download/papers/Berg10f-Spy.pdf. doi:10.1016/j.cl.2011.10.002.

[18] A. Bergel, V. P. na, Increasing test coverage with hapao, Science of Computer Programming 79 (2012) 86–100. doi:10.1016/j.scico.2012.04.006.

[19] S. Costiou, V. Aranega, M. Denker, Sub-method, partial behavioral reflection with reflectivity: Looking back on 10 years of use, The Art, Science, and Engineering of Programming 4 (2020). doi:10.22152/programming-journal.org/2020/4/5.

[20] M. Denker, O. Greevy, M. Lanza, Higher abstractions for dynamic analysis, in: 2nd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006), 2006, pp. 32–38.

[21] M. Denker, S. Ducasse, A. Lienhard, P. Marschall, Sub-method reflection, in: Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007, volume 6/9, ETH, 2007, pp. 231–251. doi:10.5381/jot.2007.6.9.a14.

[22] M. Denker, Sub-method Structural and Behavioral Reflection, PhD thesis, University of Bern, 2008.

[23] G. Kiczales, L. Rodriguez, Efficient method dispatch in PCL, in: Proceedings of ACM conference on Lisp and Functional Programming, Nice, 1990, pp. 99–105.