JOURNAL OF OBJECT TECHNOLOGY
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Object Miners: Acquire, Capture and Replay Objects to Track Elusive Bugs

Steven Costiou[a] Mickaël Kerboeuf[b] Clotilde Toullec[a]
Alain Plantec[b] Stéphane Ducasse[a]

a. Inria, Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, F-59000 Lille, France

b. Univ. Bretagne-Occidentale, UMR CNRS 6285, Lab-STICC, F-29200 Brest, France

Abstract Elusive bugs are difficult to observe and to reproduce. They are often caused by non-deterministic or unexpected events, inputs or computations. To track elusive bugs, it is necessary to narrow down the scope of the bug investigation. This helps reproducing and observing very specific aspects of the program's state and behavior. In object-oriented programs, it often comes down to finding and debugging *specific* objects. However, some objects are particularly hard to find. Typical hard cases are finding a temporary object or a single particular instance of a given class. This is the case when debugging UI elements, or programs with non-deterministic state. This capability of identifying objects of interest is crucial to the debugging of objects. Yet, debuggers addressing this problem only provide manual or limited ways to find such objects.

In this paper we present *Object Miners*: a non-intrusive object-centric debugging approach for acquiring, capturing and replaying objects. We show how the miners acquire objects at run time from the sub-results of an instrumented expression. Miners capture objects with their execution context and replay them to freeze strategic parts of the execution, eliminating non-determinism. We present a Pharo implementation of *Object Miners* along with a performance and memory overhead evaluation. We present a debugger built on top of *Object Miners*, and demonstrate through a series of examples how the application of *Object Miners* facilitates the tracking of elusive bugs. These examples include the fixing of a non-deterministic bug in an IOT application, tooling support for object-centric debugging, and the tracking of a bug in a real-world program.

1 Introduction

Debugging *elusive* bugs is an underestimated challenge. An elusive bug is difficult to reproduce, to observe and thus difficult to understand and to fix. Those bugs elude developers and often manifest themselves in symptoms distant from their root cause [Eis97].

They are due to non-deterministic computation [Zel09] or unpredictable or unforeseen program behavior [Knu89]. This is the case of bugs which only appear in production environments [Aga02, GT07, MPGB18]. The literature reports elusive bugs under different forms. *Stealth-bugs* [Eis97] are observable only after other computations have consumed their source. Causes of the bug are hard to track, as faulty objects are no longer available. *Surprise scenarios* [Knu89] are dormant bugs in the program and developers do not even suspect their source. These unexpected bugs are often due to an unforeseen or misunderstood behavior of the program. Non-deterministic inputs or computations add to the difficulty, as their reproduction cannot be guaranteed [SB17, Sch17, GT07]. Other bugs known as *Mandelbugs* [GT07] only appear after a non-deterministic period of time. Developers must not miss these bugs when they happen, or they will have to wait for their next occurrence.

These bugs are hard to reproduce outside of the context in which they appear. The traditional *stop-instrument-restart* debugging cycle [Aga02, Zel09, Spi18] hinders the reproduction of elusive bugs as it results in the loss of execution contexts. Therefore, to understand elusive bugs we need systematic means to reproduce their manifestation contexts.

In object-oriented programs, the root cause of a bug can hide inside one single object among many instances of the same class [LHS99, HJJ93]. In addition to coping with non-deterministic aspects and unpredictable behaviors, we have to narrow down the scope of the debugging investigation to the level of objects. Object-centric debugging in general allows developers to debug one specific instance without affecting other instances of the same class. Object-centric breakpoints break the execution when one chosen instance receives a particular message or when its state is modified [LHS99, RBN12, Cor16]. Object-centric instrumentation changes the behavior of specific instances through code adaptation [HJJ93, RC02, Kee04, RDT08, Tai17].

While object-centric debugging is appealing, it does not offer any practical means to *obtain* objects for instrumentation or observation. Objects must be found manually either through program inspection using breakpoints [RBN12, Cor16] or through manual selection in a dedicated graphical tool [KC03, Tai17, FLHT15]. In practice, some objects are difficult to acquire: objects stored in temporary variables, objects returned by a particular execution of a given expression, sub-results of that expression's execution, or objects whose existence in time and space is non-deterministic.

In this paper we present *Object Miners*, a complementary approach to object-centric debugging. An object miner instruments a target expression to *acquire* and *capture* objects returned by the execution of that expression and its sub-expressions. Captured objects are recorded following different strategies (reference or copy) to be used afterwards. User-defined code specifies capture conditions or post-capture actions. Captured objects from a particular (sub-)expression execution can be set as *replay objects*. Future executions of that (sub-)expression will then systematically return the replay object and remove uncertainty due to non-deterministic computations. In addition, the instrumentation is non-intrusive in the sense that it does not rewrite the original source code, nor pose design constraints. The solution and its implementation presented in this paper are currently limited to non-concurrent programs.

Object Miners help in the investigation of elusive bugs by providing systematic ways to obtain and replay specific objects for debugging. The approach also adds new support for object-centric debuggers to automatically apply object-centric instrumentation (*e.g.*, breakpoints) to captured objects. Therefore, in this paper, we present the following contributions:

1. We identify specific cases of objects of interest which are difficult to obtain using conventional debugging tools in the context of elusive bugs.

2. We define the *Object Miners* approach as a dynamic, non-intrusive instrumentation of the abstract syntax tree representation of target expressions.

3. We demonstrate the feasibility and the benefits of the approach for the debugging of elusive bugs. We present a Pharo [BDN+09] implementation of *Object Miners* with a debugger.

In the following, Section 2 describes the difficulties of finding specific objects for debugging. We illustrate the impracticability of traditional debugging tools through an example of an elusive bug in a IOT application. Section 3 presents the *Object Miners* approach, its definition, its API and its debugger, whose application is illustrated on a series of examples in Section 4. An implementation is presented in Section 5. Section 6 presents an evaluation of our implementation regarding performance and memory consumption We study related work in Section 7 and conclude in Section 8.

2 Challenge: tracking objects to debug elusive bugs

In this section, we present a motivating example illustrating the challenges that developers face when confronted with elusive bugs. We summarize the limitations of breakpoints in regards to this motivation, then we describe the difficulties of identifying objects to debug.

2.1 Motivating example: The *Sensor Monitoring App*

Let us consider the *Sensor Monitoring App* [MBC+17], a single-threaded Pharo *IOT* application deployed on a Raspberry Pi[1], in which there is an unpredictable sensor reading bug. This version of the application uses four temperature sensors to monitor the temperature of the environment where the device is deployed. Sensors 1 and 2 monitor the temperature of one environment, while sensors 3 and 4 monitor another environment. Temperatures measured for the same environment vary slightly, due to the imprecision of sensors. Figure 1 shows two screenshots of the application taken at different moments. We observe that the temperature measured by sensor 1 drops from about 32 degrees to 0, while sensor 2 continuously measures a temperature close to 32 degrees. This problem happens randomly from time to time, and cannot be manually reproduced.

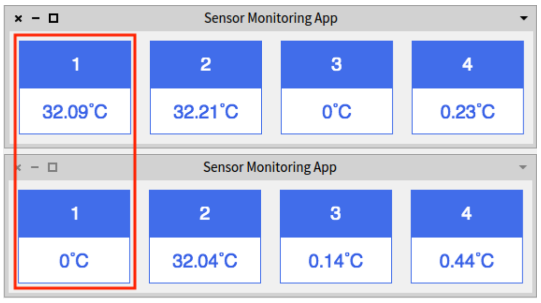


Figure 1 – Abnormal sensor reading fluctuation in the *Sensor Monitoring App*. The different windows show sensor readings at two different times. Sensor 1 view shows a temperature randomly dropping from ∼32 to 0.

Excerpts of application code are shown in Listing 1. The UI uses an instance of SensorMonitor class to query sensor data and display it (Figure 1). This is done in method sensorSweep (line 1-2) where a loop iterates over the sensors. Sensor objects are instances of class GenericSensor, which models a physical sensor. The readTemperature method (line 4) performs a

[1]https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

sensor reading (self basicRead) which returns an object encapsulating low-level data recovered from the sensor. That sensor reading object is passed to the computeTemperature: method (line 7-11) which computes the temperature from low-level data according to the sensor specifications. A guard prevents erratic values to be considered in the computation (line 8). The returned temperature is the maximum between 0 and the computed temperature, using the max: method (line 5)[2].

```
1  SensorMonitor >> sensorSweep
2    ↑ sensors collect: [ :sensor | sensor readTemperature printString ]
3
4  GenericSensor >> readTemperature
5    ↑ ((self computeTemperature: self basicRead) max: 0) round: 2
6
7  GenericSensor >> computeTemperature: sensorReading
8    sensorReading rawValue = Float nan
9      ifTrue: [ ↑ SensorReadingNaNErrorValue new ].
10   ↑ sensorReading rawValue * sensorReading maxVoltage
11     / sensorReading maxAnalogValue + sensorReading analogValueDelta
```

Listing 1 – Sensor Monitoring App simplified sensor model[3]

It seems the max: method returns 0 for some of the sensor readings, but we do not know which. It could be a sensor reading of 0, but this is suspicious because sensor 2 measures a temperature of $\sim$32 ^{o}C in the same environment (Figure 1).

The problem is that the max: method is called in an expression (line 5) whose execution uses intermediate objects computed from the execution of sub-expressions. Those intermediate objects are pushed on the context value stack[4] when they are created and popped off when they are used. This is what happens when line 5 is executed when stepping in a debugger:

1. self basicRead is executed and returns a sensor reading object that is pushed on the stack.
2. self computeTemperature: self basicRead is executed, and uses as argument the sensor object created at `1.`, that is popped from the stack. The result is pushed on the stack.
3. The literal 0 is pushed on the stack.
4. (self computeTemperature: self basicRead) max: 0) is executed, using as arguments the object created at `2.` and the literal 0. Those objects are popped from the stack.

When the execution of the max: method (point `4.`) returns an erratic value, all intermediate objects are no longer accessible because they have been popped from the stack. In addition,

[2]The max: method returns the maximum value between the receiver and the parameter of the method.

[3]For readers unfamiliar with the Pharo/Smalltalk syntax, a comparison with Java-like syntax:

- The message-send notation uses spaces instead of dots:
  sensor readTemperature printString is equivalent to sensor.readTemperature().printString().
- Arguments are specified by colons, without parenthesis:
  self computeTemperature: 0 is equivalent to this.computeTemperature(0).
- The up arrow ↑ is the return keyword: ↑ sensorReading is equivalent to return sensorReading;.
- Square brackets [ ] delimit lexical closures.

[4]In other languages, a context might be named a frame. The value stack is used for operands or methods' return value. *E.g.*, in Java, its equivalent is the frame operand stack.

one of the intermediate objects is returned from a non-deterministic computation (the sensor reading). Therefore the troublesome value cannot be reproduced on demand.

This non-deterministic aspect and the impossibility to access all related contextual data make this bug elusive.

2.2 Limitations of traditional tools to track objects

In the following, we identify limitations of logging and breakpoints regarding elusive bugs such as the one described in Section 2.1. These limitations make it hard to track objects, either because breakpoints and logging are impractical or difficult to use. We illustrate these limitations with the example *Sensor Monitoring App* described in Section 2.1.

Impracticality of breakpoints within loops. When put within a loop, a breakpoint halts the execution at each loop iteration. This is impractical if the loop has many iterations. We have to manually proceed the execution until the program stops in a context of interest for the on-going bug investigation. This is the case for methods called from within the loop in Listing 1 (line 2). Conditional breakpoints solve this problem if conditions for halting in the right context are known, and if these conditions can be expressed.

Difficulty of expressing conditions. Expressing conditions on an object with a complex state is hard and error prone. This leads to false positives, *i.e.*, inconvenient execution breaks or logging noise due to contexts or objects matching imprecise conditions. For instance, it is possible to halt in the max: method (called at line 5 of Listing 1) when it is about to return 0. But this method implements a mathematical function used by lots of different objects (*e.g.*, graphical objects). It will halt every time that condition is satisfied: for other sensor instances and for other call sites such as in the application's GUI.

Tracking modifications of an object's state. While tracking a bug, it is sometimes interesting to track modifications of an object's instance variable. We can put a breakpoint at every site in the source code where that variable is modified. We also need a condition to check if the current object modifying the variable is the object we are monitoring. To ease this process, we can use the object-centric haltOnWrite [RBN12] breakpoint, which breaks the execution when a particular state of a target object is written. However, we need to find that target object first before applying the haltOnWrite breakpoint.

Non-determinism and loss of context. We can observe the symptoms of a bug with logging or by halting the execution. However, if the source of the bug is a non-deterministic computation, its reproduction cannot be guaranteed. This is the case in Listing 1 (line 5) where a computation uses the result of a sensor reading. There is no basis to refine logging or breakpoints' conditions to narrow down the observation scope of the bug. Debuggers must provide systematic ways to cope with non-deterministic computations.

2.3 Difficulties of tracking objects of interest

Some objects are particularly hard to track when investigating elusive bugs. These objects are difficult to access from outside their context and are inaccessible when this context is passed.

Temporary objects. Temporary objects are tedious to track because they only exist for a very limited amount of time. Typically, such objects are referenced by temporary variables, objects instantiated within a local scope, and instance variables of these latter objects.

Anonymous objects. Anonymous (or non-aliased) objects are returned from a method and immediately used in another computation without being stored. Such objects are operands or method calls return value stored on a value (or operand) stack. When they are used, they are popped off that stack and references to those objects are lost in the current execution scope. This is typical of objects created or modified within expressions composed of multiple sub-expressions, such as the example depicted in Section 2.1. Developers need to track intermediate objects returned by the execution of expressions and their sub-expressions. This is a particular case of the *temporary object* problem described above.

3 Object Miners: acquire, capture and replay objects

Object Miners are instrumentation of program expressions that *capture* objects and their execution context (*i.e.*, the call stack) from those expressions and their sub-expressions. Miners access and manipulate the execution stack to capture objects. They provide access to the target expression's execution context to express capture conditions. If configured so, they *record* the captured objects from an expression and provide a history of that expression's execution. Finally, they are able to freeze the computation of a particular (sub-)expression to eliminate non-determinism by configuring an object as a *replay*.

In this section, we define Object Miners through a structural and a run-time description of the instrumentation. We illustrate the Object Miners debugger that implements the solution.

3.1 Object Miners in a nutshell

An object miner is a first-class object that instruments expressions and their sub-expressions and records the returned objects from the execution of those (sub-)expressions. Installation and configuration of object miners can be anticipated or performed *on the fly*[5] at run time. As an illustration, we revisit the code from Listing 1 with an example of object miner instrumentation.

Acquiring objects. It is the process of accessing objects during run time. Objects are *acquired* if they are accessible by a miner at the moment when they are returned by instrumented expressions and their sub-expressions.

In the following, we find the readTemperature method object (line 1) and the abstract syntax tree (AST) representing the first expression within the body of that method (line 2). We install a miner on that expression's AST (line 3): this miner will acquire all objects returned by the execution of that expression.

```
1 method := (GenericSensor lookupSelector: #readTemperature).
2 expressionAST:= method ast statements first.
3 miner := ObjectMiner reachFromAST:  expressionAST
```

Capturing objects. Object capture is the process of *recording* an acquired object. Object Miners store references to captured objects in a global variable. These objects are always accessible, during or after run time.

In the following, we configure our miner to capture objects returned by sub-expressions' execution (line 1). We set a capture condition (line 3) which is a string with source code. Within this condition we access the reified value of the acquired object.

[5] *On the fly*: when instrumentation is designed, implemented and applied without restarting the execution.

```
miner recordIntermediateObjects: true.
miner setCondition: 'value = 0'
```

Interacting with objects. Interaction with captured objects is three-fold. First, user-defined actions are executed when an object is captured. In the following, we just log the object.

```
miner setAction: 'value crLog'
```

Second, captured objects are accessible from the miner's history. Objects are stored along with the AST representing the expression from which execution they were acquired. We access below the object captured during the last execution of the instrumented expression.

```
lastCapturedObject := miner getHistory last object
```

Third, any object can be configured as a replay object for an instrumented expression (or one of its sub-expression). In the following, we configure the last captured object as a replay for the expression represented by expressionAST. Future executions of that expression are frozen, and will always return 0. Acquiring and capturing objects are disabled during replay.

```
miner replayObject: lastCapturedObject for: expressionAST
```

3.2 Structural definition of Object Miners

Instances of the ObjectMiner class define and apply *mining* instrumentation. We call them *the miners*. Miners are characterized by a miningPoint, a replayPoint, a guard and an action.

Mining point. The mining point is a language construct from which a miner will capture objects at run time. Objects are captured by harvesting the program's execution stack. A mining point is either a local variable, a method parameter, an attribute or an expression. In the case of an expression, a miner has a single mining point which captures objects returned by the evaluation of this expression. In all other cases (variables, parameters, attributes) mining points span all related assignments and reads (or message sends for attributes). *E.g.*, a miner installed on an attribute captures the object referenced by that attribute each time it is read and/or written.

Replay point. At some point in time, a miner may define one of its captured objects as a replay object. All mining points for this miner are frozen. The mining point from which the selected object was captured becomes a replay point. Each time the replay point is reached during execution, the expression or variable, assignment or parameter read/write execution is skipped. The object configured as replay is returned instead (*i.e.*, pushed on the program's execution stack), and the execution continues. Object capture for a miner in replay mode is stopped until replay is deactivated.

Guards and actions. The guard and the action are user-defined expressions. The guard is evaluated when the execution reaches a mining point and the miner is about to capture an object. If this condition evaluates to true then the object is captured. Otherwise the object is not captured and the execution continues. A guard can also be configured for a replay point. Replay is only active when the guard's conditions are satisfied or when there is no guard. Otherwise, the program executes normally. If an object is captured and an action is defined, then the action is performed just after the capture.

3.3 Record/replay at a mining point

The instrumentation applied by a miner on its mining points is composed of an abstract syntax tree (AST) representation of the mining point and a data structure to store objects captured at run time. An object miner targeting a specific expression in a method's source code is illustrated in Figure 2 and described below.

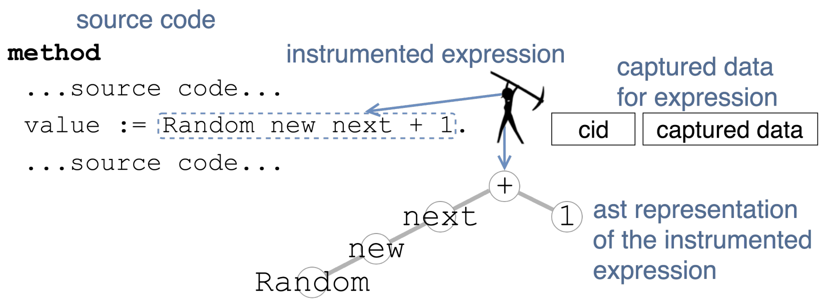

source code
method
...source code...
value := Random new next + 1.
...source code...
instrumented expression
captured data for expression
cid
captured data
+
1
next
new
Random
ast representation of the instrumented expression

Figure 2 – A miner targeting an expression (its mining point) in a method's source code. The miner works on an AST representation of its mining points, to which is associated data captured at run time. The data is a stack composed of a unique capture identifier associated to the captured object.

AST representation. The mining point is the expression: Random new next + 1 (bordered in dashes) in a method named method. Miners use the AST representation of their mining points to record from which AST node (*i.e.*, which expression) objects are captured. Each mining point owns an AST representing the instrumented entity and an associated stack that holds data structures storing the captured objects.

Data structure. Each object capture is composed of a unique capture identifier (cid) and an array of captured data. The identifier is a tag differentiating the captures. The array contains raw data with the captured object, the node from which execution it was captured, and optional data reified from the execution context.

Figure 3 illustrates a detailed object capture based on the example of Figure 2. When a mining point is executed, each sub-result is temporarily captured following the execution order in an object stack (sub-figure 1-4)[6]. When the last object is captured (sub-figure 5), corresponding to the object returned by the execution of the mining point, the *capture* algorithm is executed (sub-figure 6).

Figure 4 illustrates the replay of a captured object after some captures as depicted in the example of Figure 3. A replay is configured by selecting a captured object in the stack of captured data. This captured object becomes a *replay object*. This replay object may be a capture from any of the sub-expressions of the instrumented expression. In this illustration, the object captured from the random number generation is set as replay. When the replay is configured, the mining is temporarily deactivated and the execution of the particular sub-expression from which the object was captured is frozen. For the subsequent executions this sub-expression will always return the replay object. The miner is configured to either skip or to execute the sub-expression before replacing the returned object by that replay object.

[6] In this example, Random is a class to which the #new message is sent. The result is an instance of Random, referred to as aRandom.

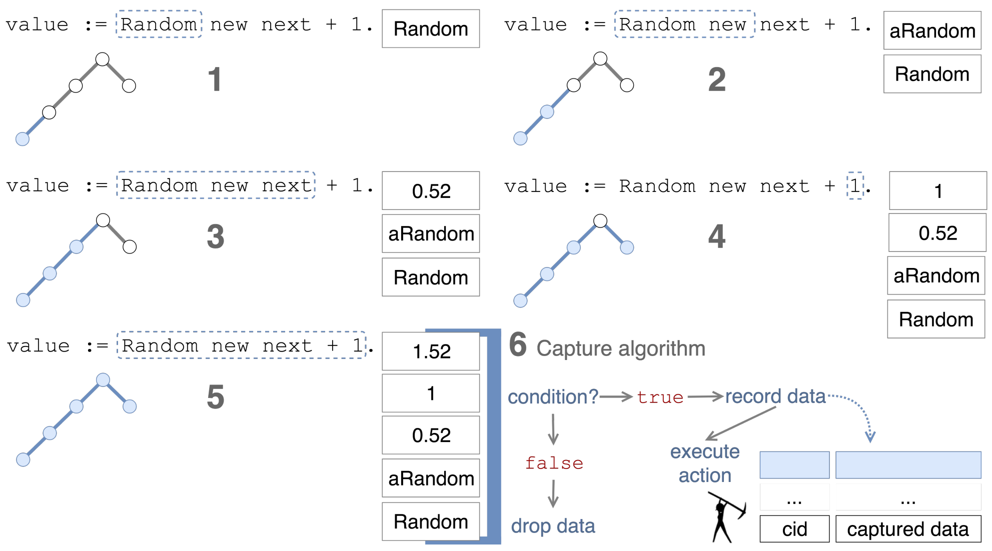


Figure 3 – Detailed object capture from an instrumented expression. Sub-results of the instrumented expression are captured in execution order (1-4). Then the object returned by the overall expression's execution is captured (5). The final capture is performed by the *capture* algorithm (6).

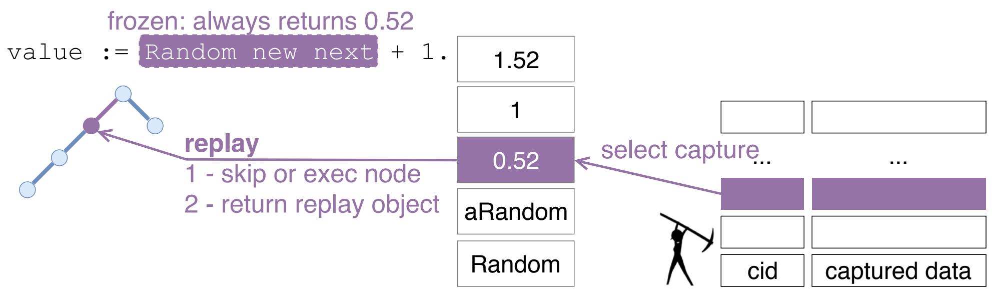


Figure 4 – Replaying a captured object from the capture stack. The object was captured from the sub-expression Random new next. The replay freezes this expression's execution. Future executions will skip the sub-expression then return the replay object (here the float 0.52).

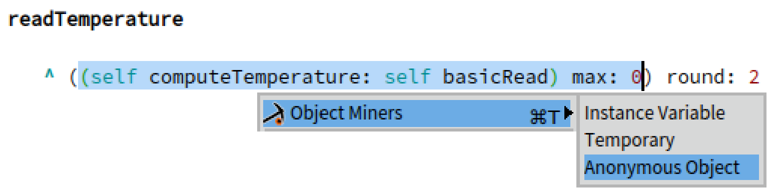
3.4 Object Miners debugger

In this section we describe the Pharo Object Miners debugger on the *Sensor Monitoring App* example, described in Section 2.1. This debugger is implemented on top of the Object Miners API[7]. This API specifies how to reach objects (*i.e.*, install miners), how to capture objects and how to interact with miners (user-defined actions, replay). It is also an implementation guide that expresses support needed to use the miners in practice (*e.g.*, to implement tools). For instance, a miner will only store raw data. The API defines an entry point to access key elements in the data structure storing captured objects. The responsibility to use the API as a specification or as a concrete implementation guide falls to the developer.

Installing a miner to capture raw sensor reading values. Miner installation is available through a contextual menu integrated in the Pharo class browser. To install a miner, we

[7]The full API is detailed online: https://github.com/ClotildeToullec/ObjectMiners

select the expression in the source code from which objects will be mined (Figure 5). Three mining options are available: mining instance variables, mining temporary variables, and mining *anonymous objects*. We select the *anonymous objects* option to mine objects from the expression returning the maximum value between 0 and the computed temperature from sensors low level readings.



```
readTemperature

    ^ ((self computeTemperature: self basicRead) max: 0) round: 2
```

Figure 5 – Contextual menu in the Pharo class browser to install object miners on the expression producing sensor reading values.

Configuration of the miner. Once a mining option has been chosen for the selected expression (Figure 5), a configuration window opens (Figure 6):

(A) We configure the recording of intermediate objects. If activated, Object Miners record results from every sub-expression's execution. In addition, we implemented record strategies that are applied to captured objects references:

- Strong reference: objects are captured by reference as defined in Section 3.
- Deep copy: miners snapshots of the entire state of captured objects. This allows for the analysis of the evolution of objects' state through multiple captures.

(B) We select information to reify at run time, which is usable in the condition and the action of the miner. The value reification represents the captured object and is always reified.

(C) We define capture conditions. Conditions are user-defined code that can refer to reifications requested in (B). If the condition is not satisfied, the capture is canceled.

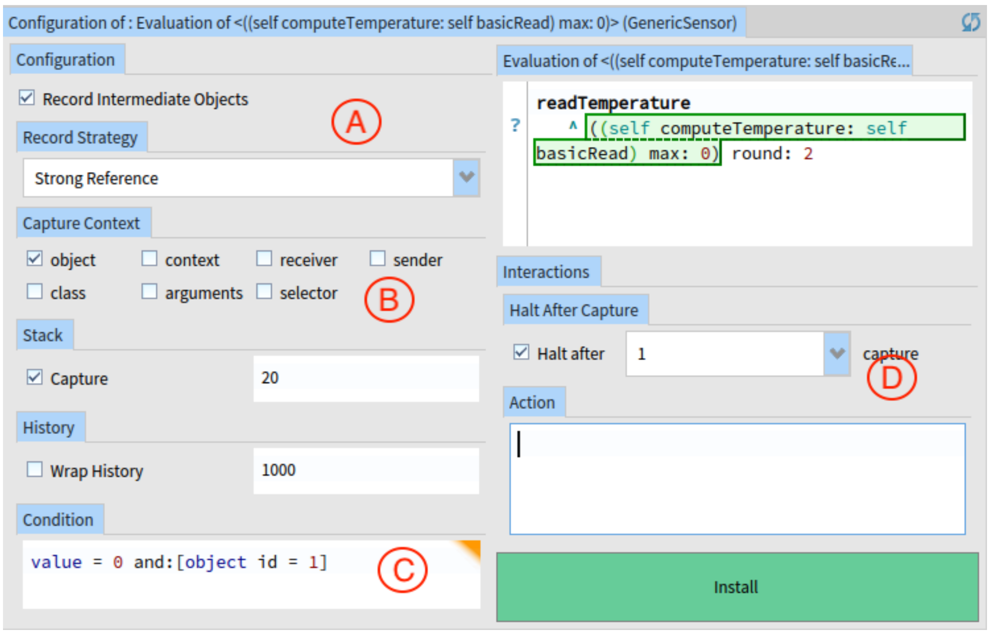
(D) A breakpoint can be configured to open a debugger after a given number of captures. This debugger will show the mining results (*i.e.*, the captured objects). The action is user-defined code that is executed after the capture, if capture conditions are satisfied.

The scope of a capture is the scope of the instrumented expression. This scope includes the local scope of the m ethod being executed and reifications provided by Object Miners (B). Available reifications[8] depend on the kind of instrumented expression (*e.g.*, an assignment, a message send...). For a given miner, capture conditions and actions execute within the scope of the expression instrumented by that miner.

Other options are available in the configuration window: the size of the captured execution stack and a history size to limit the amount of captured object.

After the miner is configured, we click on the *install* button, which applies the object miner to the target expression. This configuration can be changed and reapplied dynamically.

[8]Reifications and expressions they are available can be found at https://github.com/ClotildeToullec/ObjectMiners


Configuration of : Evaluation of <((self computeTemperature: self basicRead) max: 0)> (GenericSensor)

Configuration

☑ Record Intermediate Objects

Record Strategy: Strong Reference

Capture Context: ☑ object ☐ context ☐ receiver ☐ sender ☐ class ☐ arguments ☐ selector

Stack: ☑ Capture 20

History: ☐ Wrap History 1000

Condition:
```
value = 0 and:[object id = 1]
```

Evaluation of <((self computeTemperature: self basicRe...
```
readTemperature
    ^ ((self computeTemperature: self basicRead) max: 0) round: 2
```

Interactions

Halt After Capture: ☑ Halt after 1 capture

Action

Figure 6 – Miner configuration window: (A) recording configuration, (B) reification selection, (C) condition to capture objects, (D) break after capture. The value variable reifies the captured object.

Visualizing mining results. After the first capture, the program breaks and opens a debugger and an object miner inspector (Figure 7). It shows:

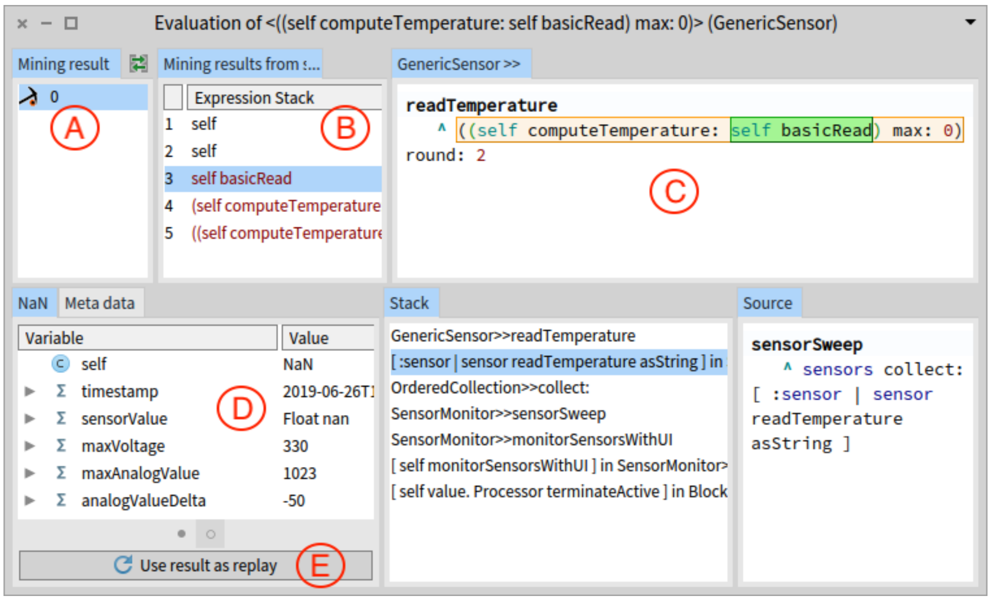
(A) The list of objects mined from the selected expression's execution (see C). This list shows a mining result from every execution of the instrumented expression. Here there was only one execution of the instrumented expression, from which the number 0 was captured.

(B) The list of sub-expressions composing the selected expression. The order of this list is the same as the execution order of the sub-expressions. Each item of this list represents a sub-expression, for which an object has been captured. For instance, we selected in the list the third expression. This expression is highlighted in green in (C) and the object mined from this expression's execution is shown in an inspector in (D).

(C) A visualization of the selected sub-expression from the list in (B). We see, highlighted in orange, the overall expression from which the object from (A) has been mined. We see, highlighted in green, the sub-expression selected from (B), and the mined object from this sub-expression (D). Just below, we see the execution stack that led to this particular execution from which the capture occured.

(D) The object mined from the sub-expression selected in (B).

(E) The *replay* button configures the object shown in (D) as a replay object for future executions of the expression from which it was mined (B-C). In this example, all future executions of the sub-expression highlighted in green (C) will return the same object (D) instead of executing the original sub-expression. Replay can dynamically be switched on and off, and the object configured as replay is interchangeable.



Mining result: 0

	Expression Stack
1	self
2	self
3	self basicRead
4	(self computeTemperature
5	((self computeTemperature

```
readTemperature
    ^ ((self computeTemperature: self basicRead) max: 0)
round: 2
```

Variable	Value
self	NaN
timestamp	2019-06-26T1
sensorValue	Float nan
maxVoltage	330
maxAnalogValue	1023
analogValueDelta	-50

```
GenericSensor>>readTemperature
[ :sensor | sensor readTemperature asString ] in
OrderedCollection>>collect:
SensorMonitor>>sensorSweep
SensorMonitor>>monitorSensorsWithUI
[ self monitorSensorsWithUI ] in SensorMonitor>
[ self value. Processor terminateActive ] in Block
```

```
sensorSweep
    ^ sensors collect:
[ :sensor | sensor
readTemperature
asString ]
```

Figure 7 – Mining results: (A) Objects mined from execution of the instrumented expression, (B) sub-expressions of the instrumented expression, (C) instrumented expression visualization, (D) object resulting from the execution of the selected (sub-)expression in (B), (E) sets the inspected object as replay for the selected (sub-)expression.

4 Debugging elusive bugs with Object Miners

In this section, we present three debugging scenarios whose investigation's ideally requires to acquire and capture objects. In these scenarios, capturing and manipulating objects of interest could provide important help while debugging. However, it is tedious to achieve such capability with traditional tools. For each scenario, we illustrate how to reach, capture and replay objects with an Object Miners debugger. We discuss how these capabilities facilitate the debugging of the presented scenarios, and we compare them with traditional tools.

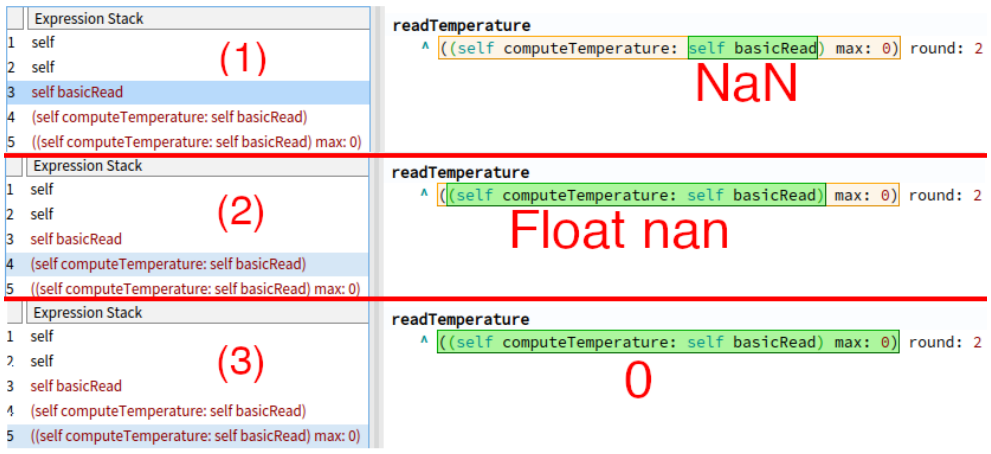
The first scenario is an example of a bug due to non-deterministic sensor readings in the *Sensor Monitoring App* [MBC$^+$17] presented in Section 2.1. We use the miners to capture sub-results of an expression and replay the problematic sensor reading to eliminate non-determinism and reproduce the bug. The second scenario is a graphical application in which we want to apply object-centric breakpoints to specific objects. We use the miners to reach objects of interest and apply object-centric breakpoints to them. Finally, the third scenario is an elusive bug encountered in a real world application named *Pillar* [ADCD16,DPD17,CTKP18]. We use the object capture and record capabilities of the miners to track and fix the bug.

4.1 Replaying objects to eliminate non-determinism

In this example we show how we use the Object Miners debugger to solve the bug in the *Sensor Monitoring App*, described in Section 2.1. We acquire and capture objects from the expression producing the inconsistent result as well as from all its sub-expressions (Listing 1). We find the erratic object and freeze the execution to replay that object and remove non-determinism. We are therefore able to reproduce the bug, understand its source and fix it. The application of Object Miners in this example is the same of the one illustrating the debugger in Section 3.4.

This bug is elusive: its non-deterministic nature makes it hard to reproduce, as sensor readings are unpredictable. Furthermore, when we see the symptoms of the bug we have already lost the objects involved in the inconsistent result. The first possible source of the bug is the low level reading of the sensor, *i.e.*, the value returned by the basicRead method. Therefore, our first step is to install a miner on the expression executing that method (Figure 5). We want to capture the values used in the expression computing the inconsistent result (*i.e.*, 0). We then configure the miner as follows (illustrated by Figure 6):

- We record intermediate objects to capture objects returned by the execution of sub-expressions.
- We select the Object reification, which represents at run-time the object executing the method readTemperature from which we mine objects. In that case, it will be an instance of GenericSensor, in which is defined that method.
- We define capture conditions: we know that sensor 1 should not read a temperature of 0. We will capture results from the overall selected expression when they are equal to 0, and if the object executing the current method is the sensor with an *id* equal to 1.
- We also define a breakpoint that will halt the execution after the first capture.



```
Expression Stack
1  self
2  self
3  self basicRead
4  (self computeTemperature: self basicRead)
5  ((self computeTemperature: self basicRead) max: 0)

readTemperature
    ^ ((self computeTemperature: self basicRead) max: 0) round: 2
```

Figure 8 – Montage of the mining results visualization. Each screenshot shows a different selected item in the expression stack and the sub-expression from which it was mined.

Tracking the bug. The inspection of sub-results from the overall instrumented expression's execution is summarized in the montage from Figure 8. It shows that the object issued from the sensor reading (Figure 8.(1), sub-expression 3) is an instance of class SensorReading [9] (Figure 7, point (D)). This object holds different values from the physical sensor, among them a sensor value which is the float nan. The sensor reading itself is displayed as NaN object, *i.e.*, *Not a Number*. It means the sensor failed to perform a physical reading. This object is passed to the computeTemperature: method, which returns the float nan (Figure 8.(2), sub-expression 4). The nan object always returns false when compared to a number[10]. Therefore the result of the max: operator between nan and 0 is always 0 (Figure 8.(3), sub-expression 5).

[9]That class name is not visible in Figure 7, we do not show the application model for the sake of readability.
[10]https://en.wikipedia.org/wiki/NaN#Comparison_with_NaN

This seems inconsistent with the code of the computeTemperature: method shown in Listing 2 (extracted from Listing 1). This case should return an error (line 2-3) instead of continuing the computation and returning the nan float.

```
1 Class GenericSensor >> computeTemperature: sensorReading
2   sensorReading rawValue = Float nan
3     ifTrue: [ ↑ SensorReadingNaNErrorValue new ].
4   ↑ sensorReading rawValue * sensorReading maxVoltage
5     / sensorReading maxAnalogValue + sensorReading analogValueDelta
```

Listing 2 – Method computing the temperature from a SensorReading instance

To reproduce the bug, we configured the NaN sensor reading object as a replay for the sub-expression from which it originated (Figure 8.(1), sub-expression 3). Therefore we can re-execute the program and guarantee that the bug will be reproduced. In the Pharo debugger that opened after the capture, we restart the execution of the readTemperature method and we start manual stepping. When we step over the sub-expression highlighted in green (Figure 7 and Figure 8.(1)), the erratic sensor reading is replayed. When we step into the computeTemperature: method (Listing 2 lines 2-3), we observe that the comparison of the sensor reading's value and Float nan returns false. Indeed, the comparison of nan with any other object, even nan, always returns false. The NaN sensor reading is then propagated to the rest of the computation. Every computation involving nan also produces nan, resulting in the method return of nan.

We fix this bug by sending the isNaN message to the value recovered from the sensor instead of comparing it with Float nan. The case is now correctly handled, and the graphical interface now displays an error message when the sensor fails to perform a physical reading.

```
Class GenericSensor >> computeTemperature: sensorReading
  sensorReading rawValue isNaN
    ifTrue: [ ... ]
```

Listing 3 – Bugfix: we use the isNaN message to know if a number is nan

Discussion. We showed how Object Miners captures objects from an expression, and how they replace future executions of one sub-expression by the replay of a captured object. Freezing specific parts of the execution with captured objects ensures bug reproduction by eliminating non-deterministic computations.

As every sub-result of the target expression is captured, we are able to see the object returned by each sub-expression execution. This gives us access to the problematic and non-deterministic object returned by the sensor reading. Without access to this sub-result, we cannot know which object produces the bug. We cannot reproduce the bug just by re-executing the code because of the non-deterministic aspect of the sensor reading. We use the replay feature to freeze the execution of the sub-expression. This systematically returns the faulty object instead of querying again the sensor by executing the original sub-expression. However, this is difficult to apply in programs with many expressions which execution produces non-deterministic buggy objects. When these expressions are spread all over the code, users have to find and instrument them all to eliminate non-determinism using the miners' replay feature.

It is possible to find and to fix the bug with conditional execution traces. However, Object Miners are, in effect, equivalent to in-memory logging. Compared to traditional logging, the miners provide facilities to express which part of the execution to record and to explore the *traces*, *i.e.*, the recorded objects, as well as to replay the objects for bug reproduction. In addition, instrumentation applied by the miners does not require to insert logging instructions in the original source code.

Conditional breakpoints are impractical to use in this case. First, we do not know which condition to express until we find the non-deterministic object. Once we know this object, we cannot easily use, *e.g.*, conditional breakpoints or unit tests to reproduce the bug. Our example is simple, yet the problematic object already has five instance variables. We do not know which of these instance variables is responsible for the bug. Therefore, we have to build up a condition (for breakpoints) or initialization code (for tests) which match the exact state of the problematic object. This is impractical and does not scale for real-world, complex objects with a large state that can have an impact on bug reproduction.

4.2 Applying object-centric breakpoints

In this example, we show how Object Miners acts as a support for object-centric debugging. We use miners to capture very specific graphical objects among thousand of them. We use a post-capture action to install an object-centric breakpoint on those objects. This provides us with a systematic and automatic means of acquiring objects to debug.

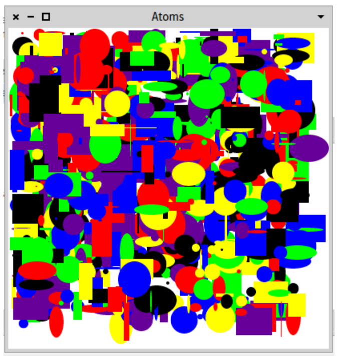


Figure 9 – The *Atom Example App*: the atom objects sometimes go offscreen instead of bouncing on the edges of the window.

The *Atom Example App* (Figure 9) is a demonstration application that we modified from the Pharo graphical framework *Bloc*[11]. It displays atoms in a window, of different shapes, sizes, colors and speeds. Atoms are instances of different classes: SquareAtom, CircleAtom, EllipseAtom. They move within the window, and bounce when they hit the edges of the window. Every atom may also teleport itself a few pixels ahead in the same direction it is moving. This happens only randomly, with a very low probability.

The bouncing bug. We introduced a bug that makes atoms randomly go off-screen instead of bouncing. This bug causes side-effects in the state of the atoms when they teleport. The faulty state prevents those atoms from bouncing correctly and they go off-screen. The correct state is only restored if an off-screen atom teleports again. In that case, they bounce normally once they come back in the visible part of the screen.

Tracking the bouncing bug. Debugging this problem is hard because it happens randomly, and it is not possible to anticipate which atom will be affected. The bouncing behavior is

[11]https://github.com/pharo-graphics/Bloc

called within the step method (Listing 4) which is itself called in a refresh loop. It iterates over the atom collection (atomsDo:), and the #bounceIn: message is sent to each atom. The bounceIn: method returns true if the atom bounced, and false otherwise. The teleport behavior is also called within the bounceIn: method. When an atom teleports, it forces the bounceIn: method to return false without actually bouncing.

We could use a conditional breakpoint in bounceIn: to halt when an atom does not bounce when it should. The problem is that each atom class implements its own bounceIn: method. Each of these methods has different return sites from which the atom is set to bounce or not. Using breakpoints is tedious as we do not know in advance which atom is going to fail bouncing. We would have to put breakpoints at each return site of every bounceIn: method of the atom class hierarchy.

```
step
  self atomsDo: [ :atom | atom bounceIn: bouncingRectangle ]
```

Listing 4 – The bouncing behavior is called at every refresh step

The haltOnCall: breakpoint [RBN12] is a powerful tool in this context. This breakpoint halts the execution when a particular method is called on a target object. All other objects are unaffected by this breakpoint. Therefore, we propose the installation of a haltOnCall:[12] breakpoint on the bounceIn: method of every atom failing to bounce. Next time an atom outside the window's edges calls that method, the execution will break. However, we need a systematic and automatic way of obtaining the failing atoms to which the breakpoint applies.

In the class browser, we look for the step method (Listing 4) and we select the #bounceIn: message send in the code. We install a miner on that expression through the contextual menu, and we configure the miner as follows:

- We activate the recording of intermediate objects, to capture objects returned by the execution of all sub-expressions.
- We ask for reifications of the execution context: the *receiver* of the message send (the atom) and the *arguments* of the message send (the bouncing rectangle).
- We set capture conditions. We capture only if:
  - the boolean returned by bounceIn: is false (*i.e.*, the atom did not bounce)
  - the atom (receiver) is outside of the bouncing rectangle (the argument)
- Capture action: we send the message haltOnCall: to the receiver of the message, *i.e.*, the captured atom. The code of this action is the following: receiver haltOnCall: #bounceIn:.

The next atoms which do not bounce and are outside the bouncing rectangle are captured. The object-centric breakpoint is automatically installed on those objects. The execution halts the next time one of those atoms fails to bounce. From there, it is easy to see that this atom has an erratic state and developers can focus the debugging investigation on this faulty object.

Discussion. Object Miners provides a systematic and automatic means of acquiring objects and their execution context. In this example we capture a returned value (the bouncing boolean) and if a condition is satisfied we apply a breakpoint on a contextual element, *i.e.*, the receiver of the message send (the atom). This facilitates the application and automation of object-centric breakpoints. In contrast, object-centric breakpoints [RBN12] originally rely only on manual

[12]We use a modern Pharo 7 implementation of this breakpoint.

selection of objects through a halted program. This is impractical in loops where the program may halt many times until an object of interest is reached.

An alternative is the use of conditional breakpoints: the program breaks when the conditions to find a buggy object are satisfied. The object-centric breakpoint is then manually applied to that buggy object. This is limited to debugging environments supporting breakpoint installation at the (sub-)expression level. Breakpoints with broader granularity (*e.g.*, line breakpoints) cannot halt the program just after a particular sub-expression has been executed and has returned an object of interest. In contrast, Object Miners supports conditional capturing of objects resulting from specific sub-expressions. If needed, developers may rewrite capture conditions and debugging actions dynamically without modifying the program's source code.

Another possibility is to manually rewrite the source code. Such rewriting implies insertion of temporary variables to store every sub-expression's return value. Then, code instrumentation using these temporary variables is inserted in the source code to determine if an object is of interest, and the developer can apply an object-centric breakpoint accordingly.

Conditional breakpoints and source code rewriting are tedious to implement when there are multiple sites of interest in the source code. This is the case for the bounceIn: methods, implemented in different atom classes, and in which there are more than one return instruction.

4.3 A real-world example: the *Pillar bug*

In the following, we relate how we solved a real bug in an open source tool by analyzing the history of captured objects and their associated captured execution stacks. The bug we present here was solved using an early prototype of the Object Miners debugger [CTKP18].

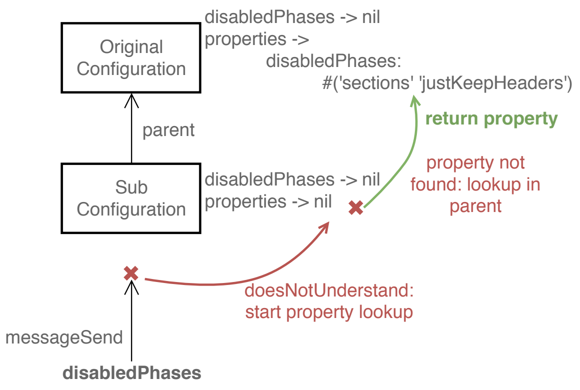


Figure 10 – Lookup of a disabledPhases property in a configuration.

4.3.1 The Pillar bug

Pillar [ADCD16] is a *mark-up* type document generator implemented in Pharo. The tool uses configuration objects composed of sub-configurations (Figure 10). They have properties which are stored in a dictionary and accessed by name through unimplemented accessors. When an unimplemented accessor message is received by a configuration, the lookup fails and an instrumentation performs an alternative lookup. That lookup checks the property dictionary of the configuration object. If the property is not found, the property lookup continues in the parent configuration until it finds the property or returns nil.

The *Pillar bug*[13] appears after a refactoring aiming at removing the alternative lookup by implementing accessors in the configuration class. For each property, an instance variable was

[13]https://github.com/guillep/pillar-bug

introduced and its accessors were implemented. These properties were correctly initialized using their accessors. When a property was requested, configurations returned the value of their corresponding instance variable instead of performing an alternative lookup.

However, the refactoring of a property named disabledPhases produced a bug in a unit test. The refactoring produced an unexpected side effect, illustrated in Figure 11. Developers forgot to copy the value of the instance variable holding the property value from the parent to the child configuration. Before refactoring, this would not be a problem as properties were automatically looked up in parent configurations. However, after refactoring, sub-configurations directly returned the value of their uninitialized disabledPhases instance variable.

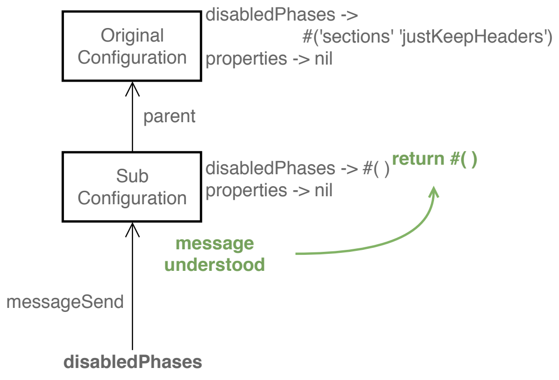


Figure 11 – Regression in the property access after refactoring: accessing the disabledPhases property in a sub-configuration returns the unitialized disabledPhases state.

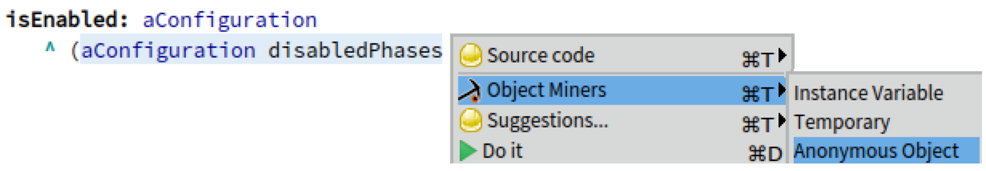
This bug completely eluded developers. Symptoms in the failing test seemed completely unrelated to the refactoring. There was a gap between the setting of configurations' properties and the resulting compiled Pillar document that was subject to assertions in the unit test.

Observing configurations through breakpoints always showed state that seemed correctly initialized. Exploring the stack from the exception showed absolutely no clue about what the problem was or how it was related to the modification. Prior to our investigation, this bug remained unsolved [CTKP18].

4.3.2 Tracking the bug with the Object Miners debugger

We were looking for the reason why removing the property lookup for the disabledPhases property by introducing a variable and its accessors produced an error when executing one of the unit tests.

Observing the erratic property value. We first wanted to see if there was a difference in the property value before (accessors are not implemented) and after refactoring (accessors are implemented). We installed a miner on the single existing message send accessing the disabledPhases property in the source code to capture objects returned by the execution of that message send (Figure 12).



```
isEnabled: aConfiguration
    ^ (aConfiguration disabledPhases
```

Figure 12 – Installation of a miner on the disabledPhases message send.

Before introducing the disabledPhases instance variable, we executed all tests and the miner captured objects from the execution of instrumented expression. All captured objects were an array containing the same two symbols, and all tests were successful.

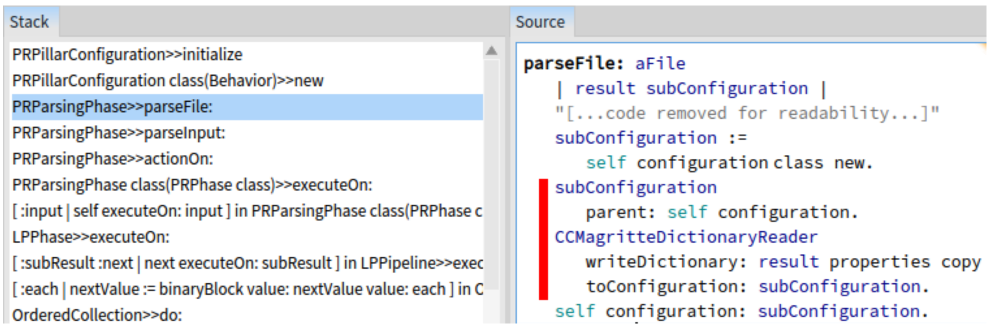
After introducing the instance variable, we executed the tests again and one of them failed. This time, the miner captured exactly the same objects except for one, which was an empty array. This was the only difference with the successful execution.

We also installed a miner on the disabledPhases instance variable to capture objects read and written in that variable. The capture history showed the variable was written only once and that the stored object was the correct array. However, the variable was read many times but one of the captured objects from a reading was the empty array. This meant at least one configuration has an uninitialized property value.

Finding the problematic configuration. We reconfigured the miner so that each time the faulty property was read or written in a configuration, that configuration was captured along with the property value. We also extended our debugger to perform deep copies of captured objects. The objective was to see there was any configuration captured more that once and if the property of that configuration was changed during the execution and put in an erratic state. However, we found that all captured configurations were different objects, and they were all captured only once when their property was read or written. This meant that configurations with a faulty state were not modified, and that they were wrong from the start.

Capturing configurations at instantiation time. We decided to capture every configuration at the time they were created to observe their initial state. We put a miner on the self variable in the initialize method of the configuration class, as this method is called each time a new instance is created. We configured this new miner to reify the execution stack at capture time, to observe the calling code. After another failing test execution, we observed that only one configuration had the correct disabledPhases array. All others had an incorrect empty array.

Analyze of the captured stacks. By analyzing the captured execution stacks (Figure 13) we found that correct and faulty configurations were not created from the same execution paths. In fact, the faulty configurations were sub-configurations all instantiated deep within the Pillar mechanics. When a sub-configuration was created before the refactoring, the properties dictionary was copied down from its parent configuration. However, the refactoring moved the disabledPhases property into an instance variable. This was our bug: after refactoring, the property value was not copied from the parent configuration into the sub-configuration's instance variable. To fix it, we inserted code to properly copy properties into sub-configurations.



```
PRPillarConfiguration>>initialize
PRPillarConfiguration class(Behavior)>>new
PRParsingPhase>>parseFile:
PRParsingPhase>>parseInput:
PRParsingPhase>>actionOn:
PRParsingPhase class(PRPhase class)>>executeOn:
[ :input | self executeOn: input ] in PRParsingPhase class(PRPhase c
LPPhase>>executeOn:
[ :subResult :next | next executeOn: subResult ] in LPPipeline>>exec
[ :each | nextValue := binaryBlock value: nextValue value: each ] in C
OrderedCollection>>do:
```

```
parseFile: aFile
   | result subConfiguration |
   "[...code removed for readability...]"
   subConfiguration :=
      self configuration class new.
   subConfiguration
      parent: self configuration.
   CCMagritteDictionaryReader
      writeDictionary: result properties copy
      toConfiguration: subConfiguration.
   self configuration: subConfiguration.
```

Figure 13 – Captured execution stack leading to the source of the *Pillar bug*. The property dictionary is copied in the sub-configuration but not the instance variable holding the correct property value.

4.3.3 Discussion

Object Miners provided helpful information and support to understand the *Pillar bug*:

- Through the history of captured objects from an expression's execution, we observed a broken invariant state between a healthy execution and a faulty execution.
- Mining objects from the faulty instance variable write accesses showed that it was never set to the problematic value, thus eliminating a possible bug source.
- Exploiting the capture history, the comparison of configurations returning correct values and faulty values showed that they were different instances.
- By mining configurations instantiations and capturing execution stacks, we found that problematic and correct configurations were not created from the same execution paths.
- Analysis of the captured stacks allowed us to find the problem that led to the faulty state.

In this scenario, we had to analyze and compare captured objects and execution stacks. To facilitate the analysis of captured objects, we implemented an ad-hoc comparison tool in the debugger. This tool performed deep copies of captured objects and checked if they their state changed between captures. It also compared the different captured objects between them to see if one of them was significantly different than others objects captured from the same expression. One interesting future work would be to explore means of automatically analyzing captured executions to help finding debugging points of interest.

5 Implementation

We implemented Object Miners[14] in Pharo [BDN+09] using the Reflectivity framework [Den08, CAD20]. We use Reflectivity's metalinks as a support to implement mining and replay points.

Metalinks. A metalink is an annotation of a method AST. It is an object that defines a message selector, a receiver named *meta-object*, and an optional list of arguments. At run time, when the code corresponding to the annotated AST is reached, the metalink is executed. First, the metalink reifies data from the execution context to build its argument list (if any). Then, the message corresponding to the selector is sent to the meta-object, with the previously computed argument list. The corresponding method is executed in the meta-object. Finally, each metalink defines if it will execute before, instead or after the code corresponding to the node it annotates. Reflectivity's instrumentation are dynamically applied through run-time bytecode rewriting, which preserves the program's source code [CAD20].

Implementing mining and replay points with metalinks. A mining point is implemented as a metalink installed on target AST nodes. Target nodes depend on the mining point definition. For example, for a mining point targeting an expression, the metalink is installed on the message node corresponding to the target expression. For a mining point targeting an instance variable of a class, a metalink is installed on each AST variable node corresponding to that variable (in methods defined in that class and its hierarchy).

An instance of MiningBehavior is associated to each metalink as a meta-object. A mining behavior is a strategy implementing concrete actions to perform when a mining point is reached during execution (conditions, object capture/replay, actions).

[14] https://github.com/ClotildeToullec/ObjectMiners

The mining behavior implements a capture: method, which message selector is given as selector to the metalink. The argument of that method is a list of reifications provided by (and configured in) the metalink. This list includes the object resulting from the evaluation of the instrumented expression (whose AST node is annotated by the metalink), and any additional reification requested by the user. The metalink's execution is controlled differently for capturing and replaying objects. When capturing, the metalink executes after the execution of the expression whose node is annotated, so that it is able to capture the returned object. When replaying, the metalink executes instead of the target expression, and returns the object configured as replay.

Building capture histories. When captured, objects are stored as raw data associated to the mining point from which they were captured. The raw data is an array, containing a reference to the captured object, its source AST (*i.e.*, the mining point) and additional optional references to objects requested by the user (*i.e.*, reifications). Each capture produces such an array, and in the case of multiple captures for the same mining point those arrays are stored in capture order (*i.e.*, the execution order). The raw data is only analyzed on demand, *e.g.*, to display mining results in the Object Miners debugger. Raw data analysis consists in enumerating captured objects to associate them with expressions and sub-expressions from which they were returned. This provides views of the local execution context for each capture of a mining point.

Object Miners as an instrumentation API. Developers interact with instances of the ObjectMiner class which implement the Object Miners API[15]. For each target to instrument (*e.g.*, an instance variable, a particular (sub)expression...), developers manually configure a miner to specify how to capture objects. Using this API, developers access the capture history of miners, change their mining specification and set captured objects as replays. All instantiated miners are automatically kept in a global variable so that they are always accessible to developers until they are uninstalled.

Discussion. We implementated the base features of Object Miners in Python (using a port of the Reflectivity library[16]) and in Java with AspectJ [Asp]. Those implementations are partial, and differ from each other on implementation details (*e.g.*, to work with static typing). However it shows that the solution is applicable to mainstream and statically typed languages.

The Pharo implementation grants unanticipated object mining because of its live programming environment. Object miners are (re)configurable *on the fly* without restarting the program. The same capability is achievable in other programming languages by using *Dynamic Software Update*. DSU is available for many languages, including but not limited to solutions for Java [PVH14] and Python [MDB15].

Additionally, *on-the-fly* (re)configuration of miners is not safe in concurrent programs, because metalinks installation is not [CAD20]. We could rely on DSU to implement miners installation, as DSU can safely update code in concurrent contexts [PH13, MDB15, TPF$^+$18]. For now, the impact of installing miners *on the fly* in a concurrent program is yet unknown.

6 Performance and space evaluation

Object Miners enables the capture of objects for debugging. Capturing objects in addition to the execution of the base program may induce a slowdown and the use of additional memory.

[15]https://github.com/ClotildeToullec/ObjectMiners
[16]https://github.com/StevenCostiou/reflectivipy

We evaluated, for a specific set of expressions, the impact of Object Miners on the execution time and on the number of objects in memory. This is not a detailed empirical evaluation of the Object Miners impact on runtimes. The goal is rather to provide developers with a general idea of that impact. We discuss the results of our evaluation under that perspective.

6.1 Experimental design

In the following, we describe our experimental design and how measures were performed.

6.1.1 Performance overhead evaluation

To evaluate the miners' impact on performance, we devised a set of expressions and compared their execution time with their Object Miners instrumented equivalent. These expressions must represent typical entities targeted by miners: assignments (for variables and attributes), message sends (for expressions) and message sends arguments (for parameters). We implemented three expressions to satisfy these constraints.

Expression 1 is an assignment. It stores an integer into the variable x. Simple assignments like this are optimized by the compiler.

```
"Expression 1: assignment, sets the variable x to 1"
x := 1
```

Expression 2 contains only message sends calling primitives from the virtual machine. Primitives execute faster than standard Pharo code.

```
"Expression 2: message sends, calling only primitives"
10 squared + 1
```

Expression 3 contains the message #compile: sent to an object[17] with an argument that is another message send. Compiling code from text is known to be slower than primitives.

```
"Expression 3: message sends, one with an argument. Compiling is known to be slow."
self class compile: self class sampleMethodSource
```

6.1.2 Number of objects overhead evaluation

Additional memory consumption happens because Object Miners captures objects and keeps references to them. This prevents garbage collection of those objects, and leads to additional memory usage. However, the amount of additional memory depends on the structure and the size of the objects' graph of captured objects. Therefore, instead of counting the space taken by captured objects, we counted how many instances they represent. To that end, we devised an expression that we instrumented with miners. We counted and compared the number of instances before and after the execution of the instrumented expression. The idea was to provide a glimpse of how many objects were kept in memory due to the usage of Object Miners.

Our goal was to simulate the capture of objects that are not optimized in memory by the Pharo system. For instance, expressions from Section 6.1.1 involve objects that are subject to such optimization, *e.g.*, integers and symbols. We designed expression 4 with a sub-expression from which an intermediate, non-optimized object can be captured. First, a message sent to the OMMemoryBenchmark class creates a dummy factory object. Another message sent to this factory object instantiates and returns an instance of OMMemoryBenchmark.

[17] When messages are sent to self, self represents the receiver.

```
"Expression 4: instantiates and returns a memory benchmark object through message sends"
OMMemoryBenchmark newDummyFactory newOMMemoryBenchmark
```

6.1.3 Instrumentation scenarios

We devised three scenarios that cover the extreme possible feature combinations of Object Miners. For each scenario, a miner is configured with the scenario's features, and installed[18] on each of the reference expressions. In the *Simple Instrumentation* scenario, we instrumented the reference expression to capture only objects returned from the execution of the overall expression. In the *Full instrumentation* scenario, we used all the available features of the miners. We executed a capture condition, captured intermediate results (*i.e.*, objects returned from sub-expressions), reified contextual data, and executed a user-defined action just after the capture. The condition and action only triggered the condition and action mechanisms, but did not execute real user code. Finally, the *Replay* scenario replayed a predefined object instead of executing the whole instrumented expression.

6.1.4 Measurement method

To measure performance overhead, we used the Pharo system method benchFor:. This method runs as many iterations as possible of the same piece of code in a given period of time. It then computes the average duration of one iteration of the benchmarked code. We used this average duration of one iteration as a comparative criterion. We arbitrarily chose to run this benchmark method using 10 seconds period. We performed this benchmark a first time for expressions 1, 2 and 3 without instrumentation. Then, for each scenario, we instrumented these expressions and we performed the benchmark again.

To measure the additional number of instances, we counted instances before and after the execution of expression 4. The expression was run in a loop for an arbitrary duration of 10 seconds. After the 10 seconds run and before counting, we performed a global garbage collection to avoid counting unreferenced objects. This experiment was repeated for each scenario. Counted instances were objects and intermediate objects captured from the execution of expression 4, and objects of the miners' model classes that were instantiated at each capture.

Each measurement was performed 30 times. We present the mean of those measures with a margin of error for a 95% confidence interval. For the performance evaluation (Table 1), we present the average execution time of each scenario's expression. For the instances counting (Table 2), we present the overhead due to the capture of instances by the miners.

Experiments were performed on a MacBook Pro/Catalina, 2,3 GHz Dual-Core Intel Core i5, 16 GB 2133 MHz LPDDR3. The experimental setup and the raw data are available online[19].

6.2 Impact of Object Miners on execution times

Hereafter, we analyze the execution times from Table 1 and we discuss threats to validity.

Simple and full miner instrumentation. When capturing from assignments and primitives, instrumented expressions executed from 93 times (expression 2, simple instrumentation) to more than 600 times (expression 1, full instrumentation) slower than a non-instrumented expression. This is due to the fact that assignments and primitives execute very fast compared to the Object Miners' mechanics. Although the simple instrumentation was 3 to 4 times faster than the full instrumentation, using miners to capture objects from assignments and primitives

[18] The code of miner's installation is available online https://github.com/ClotildeToullec/ObjectMinersBenchmarks
[19] https://kloum.io/costiou/miners/evaluation

Table 1 – Average time to run one execution of each scenario's expression (in nanoseconds).

Scenario ↓ Expression →	1) assignment	2) primitives	3) message sends
No instrumentation	6.17±00.19	11.07 ±00.09	3267298± 87687
Simple Instrumentation	1025.00±17.73	1029.00±16.28	3143172±103763
Full Instrumentation	4043.17±36.69	3233.00±15.25	3117820± 27265
Replay	8±0.26	13±0.36	14±0.26

was expensive. This is something developers must keep in mind when instrumenting, if performance is a strong constraint. Such slowdown may change the behavior of the program which theoretically affects non-determinism. In that way, this implementation of Object Miners is intrusive. However, this was not a limitation in the debugging scenarios we evaluated in Section 4.

Message sends that executed non-optimized and non-primitive code gave different results. The average execution speed of these scenarios was not significantly different than the non-instrumented version of the code. Non-instrumented code was much slower than the capture mechanism code of the miners. Consequently, the overhead is difficult to notice.

Replay scenario. Replaying objects executed as fast as an assignment or a primitive. Basically, the replay replaced the execution of an instrumented expression by a message send to a meta-object that just returned the object to replay. Object Miners' replay feature did not seem to slow down instrumented expressions.

Threats to validity. The precision of this analysis is limited. It does not rely on statistical models to compare results between our scenarios with the reference executions. The goal was only to show a glimpse of what impact Object Miners might have on instrumented programs.

We chose to run the benchmarked code for 10 seconds, then we computed the average execution time to run one time that code. This choice of duration was arbitrary, and results might not be applicable to programs with longer execution times. However, it highlights the possible overhead for executions with such lifespan (*e.g.*, unit tests).

The scenarios we investigated were also restricted to extreme cases, where we instrumented optimized code and code we knew to be slow. In a real program, interesting expressions might be a combination of those scenarios. There are also many possible Object Miners feature combination that may have different effects on performance. Finally, when instrumenting real programs we cannot know how much time an instrumented expression will be executed. The more an instrumented expression is executed, the more Object Miners will add overhead. This evaluation does not demonstrate the performance impact of the miners in general. It shows a worst-case evaluation for optimized code and an average case for slow code such as compiling.

The Pharo implementation of Object Miners was not designed to achieve high performance. It is based on meta-programming using AST manipulation. Recent work proposes approaches [MSD15] and debugging frameworks [VDVSH^{+}18] for fast meta-programming at the AST level. Using these work we could keep the same AST representation for the capture and replay of objects while achieving better performance.

6.3 Impact of Object Miners on the number of instances in instrumented programs

Hereafter, we analyze the number of instances overhead from Table 2 and we discuss threats to validity.

Table 2 – Average number of instances overhead when capturing objects from expression 4 in 10 seconds runs (counted instances after / counted instances before).

Scenario	Instances overhead
Reference	$\times 0.998 \pm 3.3 \times 10^{-6}$
Simple Instrumentation	$\times 69 \pm 2.1$
Full Instrumentation	$\times 73 \pm 1.4$
Replay	$\times 0.999 \pm 1.3 \times 10^{-5}$

Number of instances overhead evaluation. Replaying objects did not produce any instance overhead. The number of instances was similar to the reference non-instrumented scenario. In comparison, the other scenarios produced a lot of instances that were kept in memory. The simple instrumentation scenario produced 69 more instances and the full instrumentation scenario 73 times more instances than the reference scenario. This overhead is a concern because it will impact the memory usage of instrumented programs. Each additional instance that is kept by the miners will induce an additional use of memory. This was however a worst-case scenario, in which a different object was captured for each execution. It is possible in practice that the same object is captured multiple times. Object Miners may then hold multiple references to the same, unique object. Despite that, the total number of instances might continue to grow during the execution, until the memory is completely full. Two options allow developers to temper this problem. Using the miners API, developers can set up a breakpoint after a given number of captures, to stop the execution before too many objects are captured. The total number of captures can also be limited with a hard limit as in similar solutions [SB18]. To further reduce memory consumption, our implementation could be improved to eliminate equivalent and redundant objects [ISB13].

The full instrumentation scenario produced a bit more overhead than the simple instrumentation scenario. We could expect a larger gap, because in the full scenario we also capture intermediate objects. We think of two hypothesis to explain that. First, executing the full instrumentation was possibly much slower than the simple instrumentation.This means that the simple scenario's expression was executed more times in the benchmark time frame (10 seconds) and that more objects were captured. The other possibility is that the number of captured objects was negligible compared to the instances from the Object Miners' model classes. Those instances were created and used to store captured objects in a structured way. A deeper evaluation is necessary to obtain more precision about this point. This would be valuable information to design optimization for future Object Miners' implementations.

Threats to validity. When counting captured instances, we did not consider their object graph. It is possible that each capture object had a unique object graph, composed of many instances that should add up to the total number of objects that the Object Miners prevented to be garbage collected. It is also possible we missed objects in our counting, and that Object Miners actually produced more overhead for our scenarios. We also lack precision in the counting, as we did not measure separately objects meant to be captured and objects created by the miners mechanics. Those latter objects could be reduced through the optimization of the Object Miners' implementation. Similarly to the performance overhead evaluation, we chose to run our benchmarks on an arbitrary duration of 10 seconds. From these results we cannot predict exactly how the number of instances will grow in longer executions, even if the worst-case hypothesis is that it will ultimately fill the entire memory.

Nevertheless, and to mitigate this lack of precision, we believe that the important point is that for a small execution (10 seconds) and for the most extreme capture scenarios, the number of instances overhead was considerable. This might have a serious impact on an instrumented program depending on the size in memory of the captured instances. This must be a concern while instrumenting programs with Object Miners.

7 Related work

To the best of our knowledge we are not aware of existing solutions combining object tracking, recording and object replay for object-centric debugging. In this section, we study research and technical work which present similar features to those we presented this paper.

Capturing objects with traditional tools. Using conditional breakpoints we halt programs at strategic and specific points of their execution. From there it is possible to manually capture objects from the execution context, but there is no support beyond that in traditional debuggers. The Visual Studio debugger provides either the memory address of an object or a unique identifier for that object[20]. It is then possible to track this object and to reference it in other debugging operations (logging, conditional breakpoints). Most debuggers provide a *watch window* through which we are able to pin objects and to follow them (such as in the Visual Studio debugger[21]). Referencing an object (by memory address, identifiers or by pinning it) always requires to first halt the execution and manually reference the object. Moreover this does not prevent garbage collection; and in such a case, track of the object is lost.

However some objects are not easily accessible by hand from a debugger. Accessing sub-results of an expression requires halting the program then stepping over sub-expressions and manually access the last object put on the stack. This is tedious and error prone, and requires stepping over expressions whereas debuggers often support only a line granularity.

Manual capture from a list of objects. One technique consists of selecting objects by hand to capture them. This technique is used by frameworks for object-centric behavioral variations or by live programming environments. Chisel [KC03] provides an object store which references objects of interest by name. Objects of interest must be named by hand and this requires profiling of the application to identify those objects. LyRT [Tai17] provides a lookup table to access objects living in the system. Lively Groups [FLHT15] provides a graphical environment and a scene-graph from which objects are selected.

These solutions are limited when addressing the use-cases of Section 2.3. Objects with short lifespans (temporary objects) and objects which cannot be referenced (anonymous objects) are impossible to capture with this technique. Moreover, it does not scale when we need to capture a large number of objects.

Querying objects. Query-based solutions automatically search the object space to update collections of objects. In Lively Groups [FLHT15] developers query objects by defining and executing code snippets. The Bugloo [CS03] debugger provides a heap inspector accessible from a halted execution. Developers halt the execution then access the object graph through regular expression queries over the heap. Similarly, Fox [PNB04] is a language to query over heap snapshots from a running program. With query-based languages tools [WPN06, LFL$^+$16] or debuggers [LHS97, LHS99] developers define queries to find particular objects based on their state.

Expressing queries when problematic states are elusive or unknown is difficult. In the difficult cases of objects tracking (Section 2.3) queries cannot obtain the different intermediate

[20] https://docs.microsoft.com/en-us/visualstudio/debugger/using-breakpoints, accessed 07/09/19.

[21] https://docs.microsoft.com/en-us/visualstudio/debugger/watch-and-quickwatch-windows, accessed 07/09/19.

objects from the execution of sub-expressions composing a larger expression. Moreover these solutions provide dedicated query languages and tools which developers have to learn. In contrast, the API of Object Miners is written and used with the host language. There is no need to learn a new language, but developers still have to learn the miners API and debugger.

Following and recording objects. In a way, the Object Miners solution is a limited subset of the features provided by *back-in-time* or *omniscient* debuggers [Lew03]. These debuggers record the result of every execution in the running program, including state, objects, contexts and stacks. It gives the developer the ability to go backwards in time and explore how the state of a program evolved. It is a powerful technique to investigate elusive bugs and to understand their root cause. A few back-in-time debuggers focus on tracking and recording objects and present similarities with our work.

Object and flow-centric back-in-time debugging [LDGN06, LGN08, LFN09] tracks the origin and the flow of objects in a program's execution. This work focuses on providing answers about how an object arrived in a particular place and the history of its state and interactions. However, this debugger requires a dedicated virtual machine. In similar work [UP17] developers capture the execution of a program by creating run-time snapshots of domain model objects specified by the user. Snapshots record the evolution of these objects' state throughout the program execution. We see how tracked objects interacted with each other during an execution. However, we do not see if these objects were returned from a particular expression or its sub-expressions. In addition, most back-in-time debuggers record execution for post-mortem analysis. In contrast Object Miners enables unanticipated object capture and provides a means to interrupt execution for *on-the-fly* analysis.

Other tools provide live capture capabilities accessible from the development environment or from the debugger. Hermion [Röt10] dynamically captures information from the IDE during development. For instance, it captures and shows in the IDE what kind of objects are stored in variables of a program during execution. BITE [SB18] and RedShell [SB17, Sch17] integrate in the debugger a history of the last executed expressions of the current execution context. This history shows sub-results of each executed expression in the local context.

Replaying objects. Replaying executions from recorded data originally comes from *record and replay* debuggers [RDB99]. A few record/replay debuggers provide scoped replay features similar to the one of Object Miners. BigDebug [GIY+16] provides simulated breakpoints in big-data applications from which developers locally reproduce the results of a distant execution. DrDebug [WPP+14] records a user-specified part of the execution and only replays this part of the code during debugging. BITE [SB18] recreates a local context in the debugger by replacing the local execution results with recorded traces.

In comparison, the replay feature of Object Miners focuses on one (or more) target expression. When replay is active, the execution of that expression is skipped and a captured object is returned instead. We use it as a means to control execution either to reproduce a bug or to freeze the execution for further investigation.

8 Conclusion

Elusive bugs are particularly hard to observe and to reproduce. Object-centric debugging put the focus of debugging operations on specific objects of interest. This helps in narrowing down the scope of the debugging activity to track elusive bugs. However, existing object-centric solutions only provide limited ways to obtain objects for debugging. Furthermore, some objects are difficult to acquire for debugging, such as temporary or non-aliased objects, and sub-results of complex expressions.

In this paper we presented *Object Miners*, a complementary approach to object-centric debugging technique for acquiring, capturing and replaying objects without modifying the source code of the program. Miners are objects which instrument user-selected expressions of a program to capture the objects returned by the execution of these expressions and sub-expressions. Captured objects are replay-able for a particular (sub-)expression. This freezes part of the execution by systematically returning the captured object, thus eliminating non-determinism. Object Miners provides an automatic and systematic way to obtain objects for object-centric debugging.

We presented a Pharo implementation of Object Miners with a debugger. We demonstrated the feasibility and the usefulness of the solution on a series of examples, which illustrate how the Object Miners facilitate debugging in these specific cases. We performed an experiment to glimpse the impact of instrumenting programs with Object Miners. When instrumenting optimized code, the execution becomes significantly slower. When instrumenting non-optimized, slow code, there is no perceivable impact on the execution time. Object Miners has a significant impact on the number of instances in a running, instrumented program. Developers must keep that in mind when using Object Miners. This was not a limitation for the scenarios presented in this paper, as Object Miners are configurable to avoid such drawback.

In the future, we will generalize the solution to dynamically and statically typed languages as well as multi-threaded programs. We will explore how to automate tools to help developers finding objects of interest. We plan to conduct controlled experiments and experiment the solution on more real-world bugs. We will investigate performance and memory issues to obtain and evaluate efficient implementations of the miners.

References

[ADCD16] Thibault Arloing, Yann Dubois, Damien Cassou, and Stéphane Ducasse. Pillar: A versatile and extensible lightweight markup language. In *International Workshop on Smalltalk Technologies IWST'16*, Prague, Czech Republic, August 2016. URL: http://rmod.inria.fr/archives/papers/Arlo16a-IWST16-Pillar.pdf, doi:10.1145/2991041.2991066.

[Aga02] David J Agans. *Debugging: The 9 indispensable rules for finding even the most elusive software and hardware problems*. Amacom, 2002.

[Asp] AspectJ home page. http://eclipse.org/aspectj/. URL: http://eclipse.org/aspectj/.

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009. URL: http://rmod.inria.fr/archives/books/Blac09a-PBE1-2013-07-29.pdf.

[CAD20] Steven Costiou, Vincent Aranega, and Marcus Denker. Sub-method, partial behavioral reflection with Reflectivity: Looking back on 10 years of use. *The Art, Science, and Engineering of Programming*, 4(3), February 2020. doi:10.22152/programming-journal.org/2020/4/5.

[Cor16] Claudio Corrodi. Towards efficient object-centric debugging with declarative breakpoints. In *Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE2016, Bergen, Norway, 11-13 July*, 2016.

[CS03] Damien Ciabrini and Manuel Serrano. Bugloo: A source level debugger for scheme programs compiled into jvm bytecode. In *Proceedings of the International Lisp Conference 2003*, oct 2003. URL: http://www-sop.inria.fr/mimosa/fp/Bugloo.

[CTKP18] Steven Costiou, Clotilde Toullec, Mickaël Kerboeuf, and Alain Plantec. Back-in-time inspectors: an implementation with collectors. In *Proceedings of the 13th Edition of the International Workshop on Smalltalk Technologies*, IWST '18, New York, NY, USA, 2018. ACM. URL: https://hal.univ-brest.fr/hal-02320434.

[Den08] Marcus Denker. *Sub-method Structural and Behavioral Reflection*. PhD thesis, University of Bern, May 2008. URL: http://rmod.inria.fr/archives/phd/PhD-2008-Denker.pdf.

[DPD17] Thomas Dupriez, Guillermo Polito, and Stéphane Ducasse. Analysis and exploration for new generation debuggers. In *Proceedings of the 12th Edition of the International Workshop on Smalltalk Technologies*, IWST '17, pages 5:1–5:6, New York, NY, USA, 2017. ACM. URL: http://rmod.inria.fr/archives/workshops/Dupr17a-IWST-NewGenerationDebuggers.pdf, doi:10.1145/3139903.3139910.

[Eis97] Marc Eisenstadt. My hairiest bug war stories. *Commun. ACM*, 40(4):30–37, 1997. URL: http://doi.acm.org/10.1145/248448.248456, doi:10.1145/248448.248456.

[FLHT15] Tim Felgentreff, Jens Lincke, Robert Hirschfeld, and Lauritz Thamsen. Lively groups: shared behavior in a world of objects without classes or prototypes. In *Proceedings of the Workshop on Future Programming*, pages 15–22. ACM, 2015.

[GIY+16] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, and Miryung Kim. Bigdebug: Debugging primitives for interactive big data processing in spark. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 784–795. IEEE, 2016.

[GT07] Michael Grottke and Kishor S. Trivedi. Fighting bugs: Remove, retry, replicate, and rejuvenate. *Computer*, 40:107–109, 2007.

[HJJ93] Bob Hinkle, Vicki Jones, and Ralph E Johnson. Debugging objects. In *The Smalltalk Report*. Citeseer, 1993.

[ISB13] Alejandro Infante, Juan Pablo Sandoval, and Alexandre Bergel. Identifying equivalent objects to reduce memory consumption. *Proceedings of the 5th Edition of the International Workshop on Smalltalk Technologies*, page 73, 2013. URL: http://esug.org/data/ESUG2013/IWST/proceedings.pdf#page=73.

[KC03] John Keeney and Vinny Cahill. Chisel: A policy-driven, context-aware, dynamic adaptation framework. In *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, pages 3–14. IEEE, 2003.

[Kee04] John Keeney. *Completely unanticipated dynamic adaptation of software*. PhD thesis, Trinity College Dublin, 2004.

[Knu89] Donald E Knuth. The errors of tex. *Software: Practice and Experience*, 19(7):607–685, 1989.

[LDGN06] Adrian Lienhard, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Capturing how objects flow at runtime. In *Proceedings International Workshop on Program Comprehension through Dynamic Analysis (PCODA'06)*, pages 39–43, 2006. URL: http://rmod.inria.fr/archives/papers/Lien06aCapturingHowObjectsFlowPCODA06.pdfhttp://www.lore.ua.ac.be/Events/PCODA2006/pcoda2006proceedings.pdf.

[Lew03] Bill Lewis. Debugging backwards in time. In *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG'03)*, October 2003. URL: http://arxiv.org/abs/cs/0310016v1.

[LFL+16] Stefan Lehmann, Tim Felgentreff, Jens Lincke, Patrick Rein, and Robert Hirschfeld. Reactive object queries: consistent views in object-oriented languages. In *Companion Proceedings of the 15th International Conference on Modularity*, pages 23–28, 2016.

[LFN09] Adrian Lienhard, Julien Fierz, and Oscar Nierstrasz. Flow-centric, back-in-time debugging. In *Objects, Components, Models and Patterns, Proceedings of TOOLS Europe 2009*, volume 33 of *LNBIP*, pages 272–288. Springer-Verlag, 2009. URL: http://scg.unibe.ch/archive/papers/Lien09aCompass.pdf, doi:10.1007/978-3-642-02571-6_16.

[LGN08] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical object-oriented back-in-time debugging. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *LNCS*, pages 592–615. Springer, 2008. ECOOP distinguished paper award. URL: http://scg.unibe.ch/archive/papers/Lien08bBackInTimeDebugging.pdf, doi:10.1007/978-3-540-70592-5_25.

[LHS97] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. Query-based debugging of object-oriented programs. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming (OOPSLA'97)*, pages 304–317, New York, NY, USA, 1997. ACM. doi:10.1145/263698.263752.

[LHS99] Raimondas Lencevicius, Urs Hölzle, and Ambuj Kumar Singh. Dynamic query-based debugging. In R. Guerraoui, editor, *Proceedings of European Conference on Object-Oriented Programming (ECOOP'99)*, volume 1628 of *LNCS*, pages 135–160, Lisbon, Portugal, June 1999. Springer-Verlag.

[MBC+17] Matteo Marra, Elisa Gonzalez Boix, Steven Costiou, Mickaël Kerboeuf, Alain Plantec, Guillermo Polito, and Stéphane Ducasse. Debugging cyber-physical systems with pharo. In *Proceedings of the 12th Edition of the International Workshop on Smalltalk Technologies*, IWST '17, pages 8:1–8:10, New York, NY, USA, 2017. ACM. URL: https://hal.archives-ouvertes.fr/hal-01585349, doi:10.1145/3139903.3139913.

[MDB15] Sébastien Martinez, Fabien DAGNAT, and Jérémy Buisson. Pymoult : On-line updates for python programs. In *ICSEA 2015 : 10th International Conference on Software Engineering Advances*, pages 80 – 85, Barcelone, Spain, November 2015. URL: https://hal.archives-ouvertes.fr/hal-01247603.

[MPGB18] Matteo Marra, Guillermo Polito, and Elisa Gonzalez Boix. Out-Of-Place debugging: a debugging architecture to reduce debugging interference. *The Art, Science, and Engineering of Programming*, 3(2), November 2018. URL:

https://hal.inria.fr/hal-01952790, doi:10.22152/programming-journal.org/2019/3/3.

[MSD15] Stefan Marr, Chris Seaton, and Stéphane Ducasse. Zero-overhead metaprogramming: Reflection and metaobject protocols fast and without compromises. In *PLDI - Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15. ACM, jun 2015. URL: http://rmod.inria.fr/archives/papers/Marr15a-PLDI15-ZeroOverheadMetaprogramming.pdf, doi:10.1145/2737924.2737963.

[PH13] Luis Pina and Michael Hicks. Rubah: Efficient, general-purpose dynamic software updating for java. In *Proceedings of the 5th International Workshop on Hot Topics in Software Upgrades (HotSWUp)*, 2013.

[PNB04] Alex Potanin, James Noble, and Robert Biddle. Snapshot query-based debugging. In *Proceedings of the 2004 Australian Software Engineering Conference (ASWEC'04)*, page 251, Washington, DC, USA, 2004. IEEE Computer Society.

[PVH14] Luís Pina, Luís Veiga, and Michael Hicks. Rubah: Dsu for java on a stock jvm. *ACM SIGPLAN Notices*, 49(10):103–119, 2014.

[RBN12] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. Object-centric debugging. In *Proceeding of the 34rd international conference on Software engineering*, ICSE '12, 2012. URL: http://scg.unibe.ch/archive/papers/Ress12a-ObjectCentricDebugging.pdf, doi:10.1109/ICSE.2012.6227167.

[RC02] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of European Conference on Object-Oriented Programming*, volume 2374, pages 205–230. Springer-Verlag, 2002.

[RDB99] Michiel Ronsse and Koen De Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Transactions on Computer Systems (TOCS)*, 17(2):133–152, 1999.

[RDT08] David Röthlisberger, Marcus Denker, and Éric Tanter. Unanticipated partial behavioral reflection: Adapting applications at runtime. *Computer Languages, Systems & Structures*, 34(2-3):46–65, 2008.

[Röt10] David Röthlisberger. *Augmenting IDEs with Runtime Information for Software Maintenance*. PhD thesis, University of Bern, May 2010. URL: http://scg.unibe.ch/archive/phd/roethlisberger-phd.pdf.

[SB17] Stefan Schulz and Christoph Bockisch. Redshell: Online back-in-time debugging. In *Companion to the first International Conference on the Art, Science and Engineering of Programming*, page 1. ACM, 2017.

[SB18] Stefan Schulz and Christoph Bockisch. A blast from the past: online time-travel debugging with bite. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, page 13. ACM, 2018.

[Sch17] Stefan Schulz. Back-in-time evaluation: Towards online trace-based debugging. In *Companion to the first International Conference on the Art, Science and Engineering of Programming*, page 40. ACM, 2017.

[Spi18] Diomidis Spinellis. Modern debugging: The art of finding a needle in a haystack. *Commun. ACM*, 61(11):124–134, October 2018. doi:10.1145/3186278.

[Tai17] Nguonly Taing. *Run-time Variability with Roles*. PhD thesis, Saechsische Landesbibliothek-Staats-und Universitaetsbibliothek Dresden, 2017.

[TPF+18] Pablo Tesone, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, and Stéphane Ducasse. Dynamic software update from development to production. *Journal of Object Technology*, 2018. URL: http://rmod.inria.fr/archives/papers/Teso17b-JOT-DynamicUpdate.pdf, doi:10.5381/jot.2018.17.1.a2.

[UP17] Peter Uhnák and Robert Pergl. Ad-hoc runtime object structure visualizations with metalinks. In *Proceedings of the 12th edition of the International Workshop on Smalltalk Technologies*, page 7. ACM, 2017.

[VDVSH+18] Michael Van De Vanter, Chris Seaton, Michael Haupt, Christian Humer, and Thomas Würthinger. Fast, flexible, polyglot instrumentation support for debuggers and other tools. *The Art, Science, and Engineering of Programming*, 2(3), 2018. URL: http://dx.doi.org/10.22152/programming-journal.org/2018/2/14, doi:10.22152/programming-journal.org/2018/2/14.

[WPN06] Darren Willis, David J Pearce, and James Noble. Efficient object querying for java. In *Thomas D. (eds) ECOOP 2006 – Object-Oriented Programming. ECOOP 2006. Lecture Notes in Computer Science, vol 4067*, pages 28–49. Springer, Berlin, Heidelberg, 2006.

[WPP+14] Yan Wang, Harish Patil, Cristiano Pereira, Gregory Lueck, Rajiv Gupta, and Iulian Neamtiu. Drdebug: Deterministic replay based cyclic debugging with dynamic slicing. In *Proceedings of annual IEEE/ACM international symposium on code generation and optimization*, page 98. ACM, 2014.

[Zel09] Andreas Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

About the authors

Steven Costiou is an Inria researcher in the RMoD team. Before that, he worked six years in the industry as a software developer in various areas (defense, aerospace, point-of-sale software). He then did research on unanticipated software adaptation during his PhD. Today, he is interested in the identification and the study of the properties that programming languages and their infrastructure must exhibit to support new debugging features that effectively help debugging. This research spans different topics: meta-programming, object-centric instrumentation and dynamic software adaptation. Contact him at steven.costiou@inria.fr.

Mickaël Kerboeuf is an associate professor in the Department of Computer Science at University of Western Brittany (UBO). He is member of the Lab-STICC lab (UMR CNRS 6285), in MOCS team targeting hardware/software design methodologies and tools in the domain of embedded systems. His research pan both code reuse in the context of model driven engineering and dynamic adaptation in the context of reflective dynamic languages. Much of his work has been on formal validation of reuse and adapation. Contact him at mickael.kerboeuf@univ-brest.fr.

Clotilde Toullec is a research software engineer in the RMoD team at Inria Lille - Nord Europe. She works on software analysis and reenginering. She is developing the open-source reengineering platform Moose http://www.moosetechnology.org/. Her interests include IDE and tools development, metamodelization, reverse engineering, tests and software maintenance. Contact her at clotilde.toullec@inria.fr.

Alain Plantec is a professor in computer science at the University of West Brittany. He is also a member of the Lab-STICC lab (UMR CNRS 6285), France. His favorite domains are object oriented programming, design of interoperable software components and code generators building. His research activities mainly focus on domain specific modeling environments design for the implementation and the validation of complex systems such as real-time and safety-critical systems. Contact him at alain.plantec@univ-brest.fr.

Stéphane Ducasse I'm an Inria Research Director. I lead RMoD team http://rmod.lille.inria.fr. From 2011-2014 I have been Delegue scientific of the Inria nord Europe lab (300 researchers, 17 teams). I'm expert in language design and reengineering. I worked on Traits. Traits have been introduced in Pharo, Perl, PHP and under a variant into Scala, Groovy and Fortress. I'm expert on software quality, program understanding, program visualisations, reengineering and metamodeling. I'm one of the developer of Moose, an open-source software analysis platform http://www.moosetechnology.org/. I created Synectique a company building dedicated tools for advanced software analyses. I'm one of the leader of Pharo http://www.pharo.org/ a dynamic reflective object-oriented language supporting live programming. Since 2013, I built the industrial Pharo consortium http://consortium.pharo.org. I work regularly with companies (Thales, Wordline, Siemens, Berger-Levrault, Arolla,...) on software evolution problems. I wrote couple hundred articles and several books. According to google my h-index is 55 for more than 12800 citations. I like to help people becoming what they want and building things. Contact him at stephane.ducasse@inria.fr.

Acknowledgments We would like to thanks Christopher Fuhrman and Guillermo Polito for their careful proof-reading. This work is supported by I-Site ERC-Generator Multi project 2018-2022. We gratefully acknowledge the financial support of the Métropole Européenne de Lille.